Cours du 10 mars

Consensus, diffusion atomique, réplication

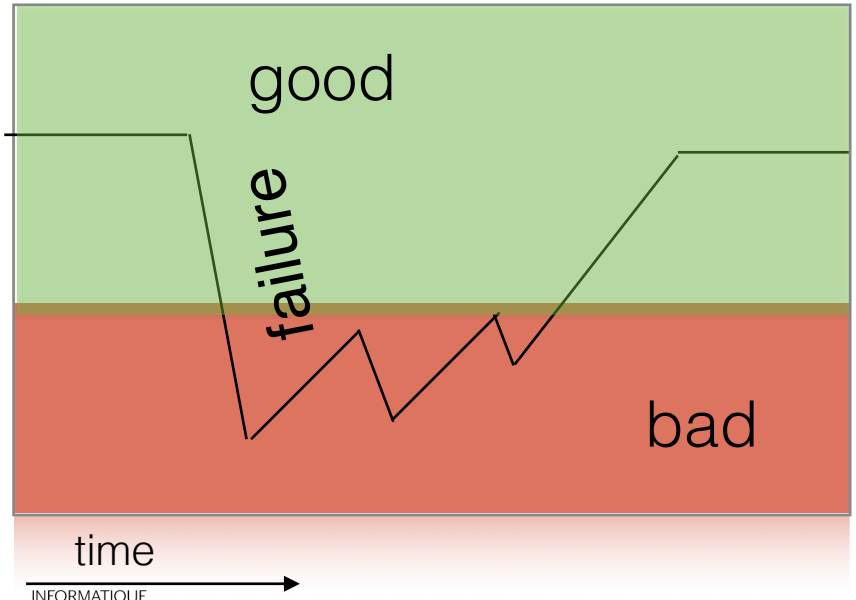
Auto-stabilisation
(Self-stabilization)

(Reprise du dernier cours)

auto-stabilisation

Partant d'un état quelconque des variables, l'algorithme finit par satisfaire sa spécification:

- Un jour l'état du système est sûr (satisfait la spécification)
- Une fois dans un état sûr, l'algorithme reste dans un état sûr



- Des défaillances transitoires (transient) comme corruption de la mémoire
- Auto-stabilisation: un jour tout redevient normal

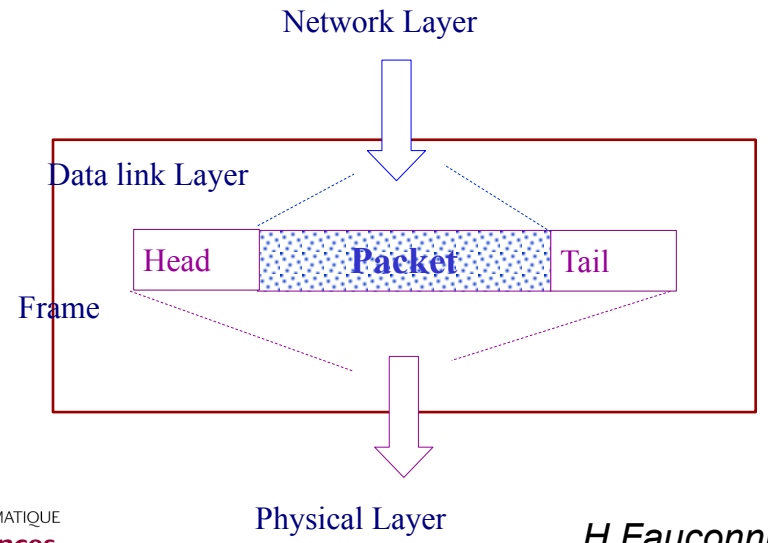
exemple: protocole du bit alterné

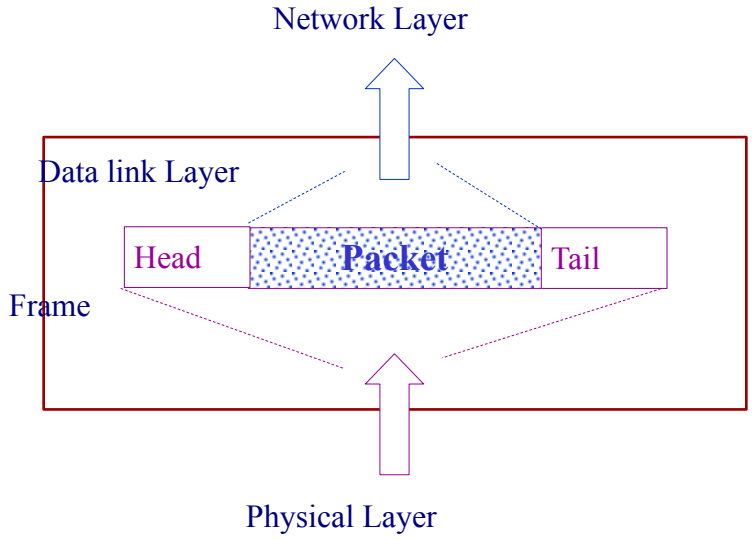
Un algorithme (classique) de la couche liaison de données:

Réaliser un transfert fiable de données sur un lien de communication non fiable:

Le lien de communication peut perdre des messages

Grâce à des retransmissions des messages l'alogithme permet d'obtenir un flux de données fiable



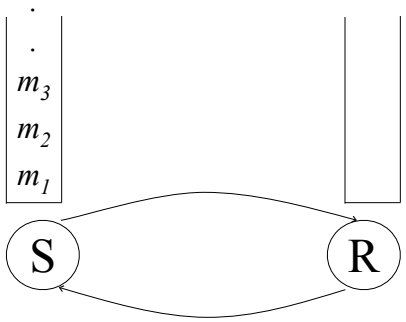


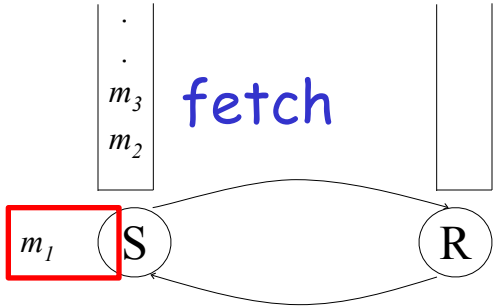
liaison de données

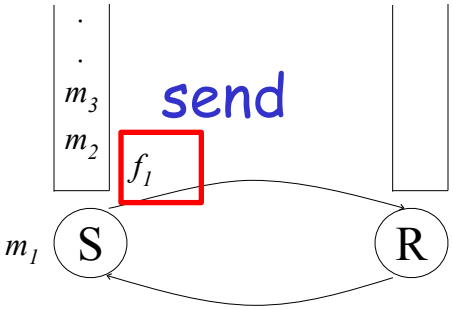
Flot des messages:

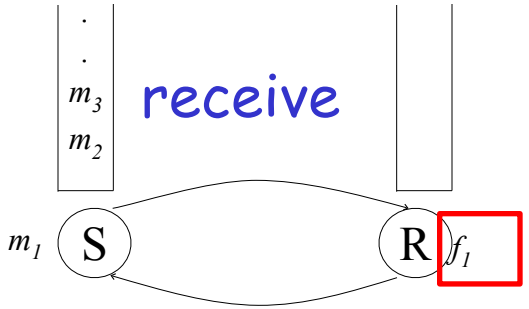
File d'entrée des messages
File de sortie des messages

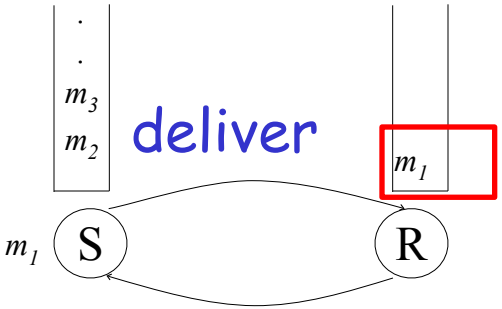
fetch-deliver
send-receive

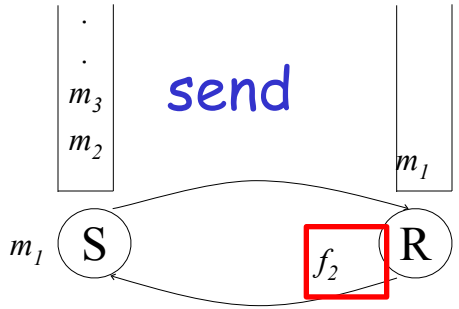












auto-stabilisation, Protocole du bit alterné

L'algorithme:

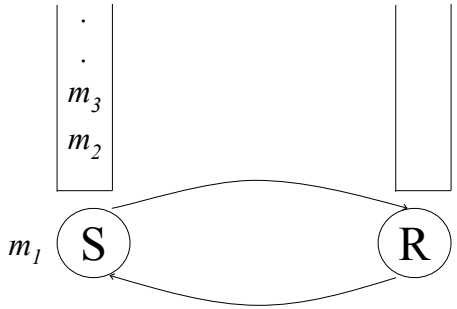
S l'émetteur a une file infinie de messages d'entrée (in[1],in[2],…) à transférer vers le récepteur dans le même ordre sans omissions ni duplications

R le récepteur a une file de messages de sortie (out[1],out[2],…). La séquence des messages de sortie dans la file doit toujours être un préfixe de la séquence des messages dans la file d'entrée.

Principes de l'algorithme:

- **Canaux**:
  - Un canal die l'émetteur vers le récepteur et un canal du récepteur vers le sender
  - Le canaux peuvent perdent de messages
- **Les processus:**
  - ajouter un bit à chaque message
  - alterner les bits: ajouter successivement 0 puis 1
  - Le récepteur accepte le message si le bit est le bit attendu (0 ou 1) et envoie un ack (0 ou 1)
  - Le récepteur envoie un nouveau message s'il a reçu le bit d'ack attendu
  - Timeout (sans réception de l'ack attendu) : l'émetteur envoie à nouveau le message

Protocole



Code pour l'émetteur s:

1 **Initialization**
2 $i := 0$
3 $bit_s := 0$
4 $send(bit_s, in[i])$
5 $reset(Timer)$
6 **TIMEOUT:**
7 { **Timeout: Timer a expired** } $\longrightarrow$
8 $send(bit_s, in[i])$
9 **SEND:**
10 { **A message** (b) **in Queue** Q_s} $\longrightarrow$
11 $receive(b)$
12 **if** $b = bit_s$ **then**
13 $bit_s := 1 - bit_s$
14 $i := i + 1$
15 $send(bit_s, in[i])$
16 $reset(Timer)$

Comportement du canal

1 **LOSS** Q_s**:**
2 $\{m \in Q_s\} \longrightarrow$
3 $Q_s := Q_s - \{m\}$
4 **LOSS** Q_r**:**
5 $\{m \in Q_r\} \longrightarrow$
6 $Q_r := Q_r - \{m\}$

Code du récepteur r:

1 **Initialization**
2 $j := 0$
3 $bit_r := 1$
4 **RECIEVE:**
5 { **A message** $(b, data)$ **in Queue** Q_r} —
6 $receive(b, data)$
7 **if** $b \neq bit_r$ **then**
8 $bit_r := 1 - bit_r$
9 $j := j + 1$
10 $out[j] := data/*Deliver*/$
11 $send(bit_r)$

Bit alterné:

Sûreté?

- Les messages de la file de sortie correspondent aux messages de la file d'entrée

Vivacité?

- Progression: i_r augmente avec le temps

Etat sûr?

- On ne considère que les bits associés aux messages: Q_r représente les bits du canal de l'émetteur vers le récepteur, Q_s représente les bits du canal du récepteur vers l'émetteur.
- Un état sûr est un état (bit_s, Q_r, bit_r, Q_s) « correct » les timeout (et le comportement du canal) peuvent dupliquer les messages.
- Les états sur correspondent à $0*1* \vee 1*0*$

Si on est dans un état sûr on reste dans un état sûr.

On peut montrer que si on reste dans des états sûrs les messages de la file de sortie correspondent aux messages de la file d'entrée:

$$\forall k(0 \leq k \leq i_r) out[k] = in[k]$$

Bit alterné

On suppose que le canal assure que:

- si une infinité de messages est émis une infinité de message est reçu.

Cette propriété assure la vivacité:

- Supposons que seulement un nombre fini k de messages est délivré: soit (b, m) ce dernier message, et $i_r = k$, dans ce cas le message $k+1$ n'est jamais délivré. Grâce aux timeout,
  - Si s envoie indéfiniment le k-ème, dans ce cas r le recevra et enverra une infinité de b qui feront que s enverra $(k+1)$-ème message une infinité de fois qui sera délivré.
  - Sinon s envoie une infinité de fois le $(k+1)$-ème message qui sera délivré

Bit alterné

(Pseudo)-stabilisation:

- Supposons que l'état ne soit pas sûr (faute transitoire de la mémoire) on peut délivrer des « mauvais » messages qui seraient dans le canal: uniquement un nombre fini d'erreur.
- (Cependant ces erreurs peuvent arriver n'importe quand. Pour une exécution donnée il existe un temps à partir duquel tout ira bien, mais il est possible qu'une erreur arrive plus tard)

Défaillances…

On considère ici des pannes « crashs »

- $\{p_1, \cdots, p_n\}$ n processus au plus t parmi eux peuvent tomber en panne (=arrêter d'exécuter leur code)
- Dans une exécution p est correct s'il fait une infinité de pas d'exécution sinon il est défaillant

Algorithme t résilient: vérifie sa spécification si au plus t parmi eux peuvent s'arrêter.

Consensus

On va considérer un problème très simple mais fondamental: le consensus

Spécification:

- algorithme de décision:
  - tous les processus p ont une valeur initiale v_p prise dans un ensemble V ($\forall p : v_p \in V$), chaque processus doit décider de façon irrévocable (écriture d'une variable d_p qui ne peut être écrite qu'une seule fois: écriture de d_p= décision de p)
  - qui vérifie
    - Accord: si p et q (corrects?) décident alors ils décident la même valeur
    - Validité: la valeur décidée est une des valeurs initiales
    - Terminaison: tout processus correct décide.

Rondes synchronisées

On considère ici un modèle synchrone pour les processus et les communication.

Rondes synchronisées

Suite de « rondes » r=1, …, m,…

- Dans la ronde r:
  - Chaque processus envoie à tous
  - Chaque processus reçoit de tous les messages de la ronde r
  - Chaque processus change son état (suivant ce qu'il a reçu)

Remarques

Exercices:

- montrer qu'on peut réaliser des rondes synchronisées si les processus et la communication sont synchrones.
- (En déduire que:
  - Il existe un algorithme pour P dans le modèle de rondes synchronisés si et seulement si il existe un algorithme pour P dans le modèle avec processus et communication synchrones)
- Que se passe-t-il en ce qui concerne les pannes des processus?

Exercice:

Essayer de trouver un algorithme de consensus:

- Dans un système synchrone sans panne $(t = 0)$
- Dans un système asynchrone sans panne $(t = 0)$
- Avec t pannes et des rondes synchronisées où si p tombe en panne dans la ronde r, soit aucun de ses messages n'arrive soit tous ses messages arrivent

Floodset

- V ensemble des valeurs initiales, t le nombre de processus qui peuvent tomber en panne
- Chaque processus p maintient $W_p \subseteq V$; initialement $W_p = \{v_p\}$
- À chaque ronde $r = 1, \cdots, t+1$ chaque processus p envoie W_p et ajoute à W_p toutes les valeurs reçues à la ronde.
- Après $t+1$ rondes: p décide $\min W_p$

Floodset

Soit $W_p(r)$ la valeur de W_p après la ronde r

- Si dans la ronde r aucun processus ne tombe en panne: pour tout p, q vivant à la ronde r $W_p(r) = W_q(r)$
- Si $W_p(r) = W_q(r)$ pour tout p, q vivants à la ronde r alors pour tout r', $r \leq r' \leq t+1$ pour tout p, q vivants à la ronde r' $W_p(r') = W_q(r')$
- pour tout p, q vivants à la ronde $t+1$ $W_p(t+1) = W_q(t+1)$

  (Par le principe des tiroirs: il y a au moins une ronde sans panne)
- Floodset résout le consensus

  Remarques:
  - On a plus que du consensus : tous les processus corrects ont le même ensemble de valeurs qui contient les valeurs de tous les processus corrects
  - C'est un protocole où toute l'information est transmise (full information protocol)

Remarques

- Est-il nécessaire de faire $t + 1$ rondes? (difficile)
- Peut-on dans certains cas terminer avant la ronde $t + 1$

(Atomique) Commit

Engagement: commit(1) /abort (0). Si au moins un processus veut abort alors tout le monde doit décider abort, si tous les processus proposent commit et qu'il n'y pas de panne alors tout le monde doit décider commit.

- Accord: si p et q décident ils décident de la même valeur
- Validité:
  - si un processus a 0 comme valeur initiale alors la seule décision possible est 0 (abort)
  - Si tous les processus ont 1 comme valeur initiale et qu'il n'y a pas de défaillance la seule décision possible est 1 (commit)
- Terminaison
  - (Faible) s'il n'y a pas de défaillance tous les processus décident
  - (Forte) tous les processus corrects décident

2-phase commit

p_1 a un statut particulier (leader)

Ronde 1:

- Tous les processus envoient leur valeur initiale à p_1 et décident 0 si leur valeur initiale est 0
- p_1 collecte les messages s'il n'a reçu que des 1 et qu'il a reçu de tous il décide 1. Dans tous les autres cas, (au moins un 0 ou pas de messages d'un processus) il décide 0

Ronde 2:

- p_1 envoie sa valeur de décision à tous qui décident cette valeur (s'ils n'ont pas déjà décidé)

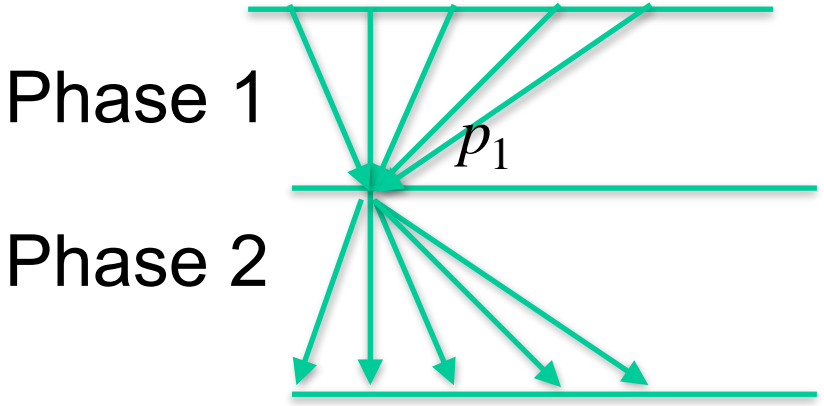
- Commit avec terminaison faible (pourquoi?)

2-phase commit

Validité?

Accord?

Terminaison?



3-phase commit

Dans le 2-phase commit p_1 peut tomber en panne ce qui empêche la terminaison. On passe à des 3-phases, dans une 3-phase, il y a un coordinateur c (au début p_1, ensuite p_2, etc.)

Ronde 1:

- Tous sauf p_1 envoient leur valeur à p_1 (et décident 0 si 0). p_1 collecte les valeurs si tous les messages sont arrivés et sont des 1, p_1 est « prêt » sinon décide 0

Ronde 2:

- Si p_1 a décidé 0 p_1 envoie 0 sinon « prêt ». Un processus qui reçoit 0 décide 0, un processus qui reçoit « prêt » devient « prêt ». Si p_1 est « prêt » il décide 1.

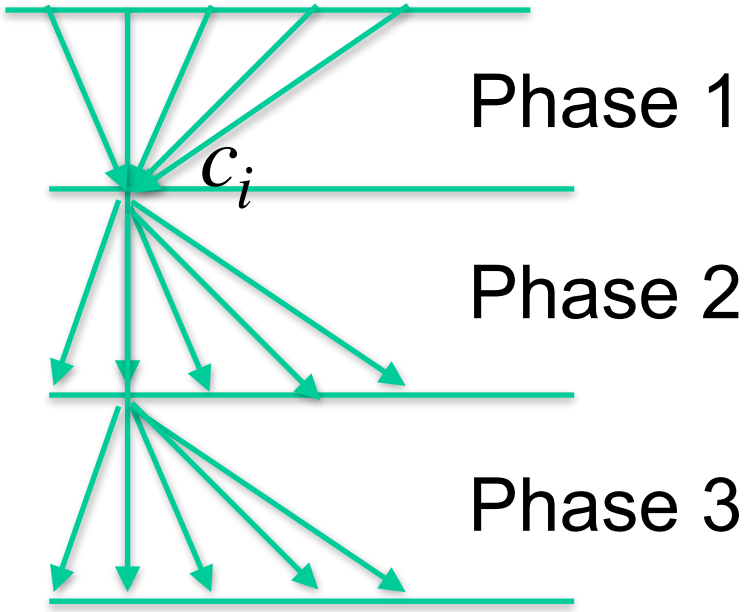
Ronde 3:

- Si p_1 a décidé 1 p_1 envoie 1, un processus qui reçoit 1 décide 1

- Après la ronde 3 les processus sont dans un des états: d_0(décision 0) d_1(décision 1) « prêt » ou « ? » (pas de décision et pas « prêt »)

États:

- d_0, d_1 décision
- « prêt »:
- « ? » incertain



3-phases commit

Après ces 3 rondes:

- Si un processus est « prêt » ou dans l'état d_1 toutes les valeurs initiales sont 1
- Si un processus est dans l'état d_0 aucun processus n'est dans l'état d_1 et aucun processus vivant dans l'état « prêt »
- Si un processus est dans l'état d_1 aucun processus n'est dans l'état d_0 et aucun processus vivant dans l'état « ?»

- L'accord, la validité sont assurés et si le processus 1 est vivant la terminaison est assurée.

3-phases commit

Ronde 4:

- Tous envoient leur valeur à p_2 (d_0, d_1, « prêt », « incertain »). p_2 collecte les valeurs
  - si d_0 parmi les valeurs et p_2 n'a pas décidé, p_2 décide 0
  - si d_1 parmi les valeurs et p_2 n'a pas décidé p_2 décide 1
  - Si au moins un « prêt » p_2 devient « prêt »
  - Si « incertain » parmi les valeurs p_2 décide 0

Ronde 5:

- p_2 se comporte comme $p1$ dans la ronde 2 et 3 dans la ronde 5 et 6: s'il a décidé il envoie sa décision, sinon il envoie « prêt » . Sur réception d'une décision de p_2, un processus décide (s'il ne l'a pas déjà fait). Sur réception de « prêt » il devient « prêt » p_2 décide 1 (s'il n'a pas décidé)

Ronde 6:

- Si p_2 a décidé 1 il envoie $decide(1)$. Un processus qui reçoit $decide(1)$ décide 1 (sil ne l'a pas déjà fait).

- Ensuite le protocole applique les rondes 4,5,6 au processus $p_3, \cdots, p_n$

Autres défaillances…

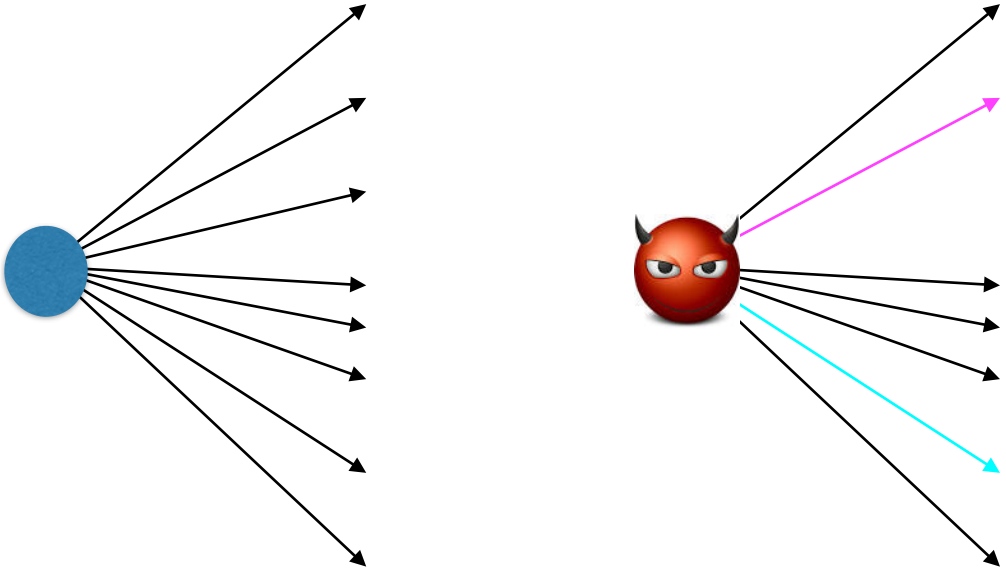
Byzantins..

défaillance byzantine : un processus byzantins ne suit pas son code et peut avoir n'importe quel comportement. Il peut par exemple envoyer n'importe quel message.

(Mais il ne peut pas se faire passer pour un autre- on suppose de plus qu'on peut savoir qu'on n'a pas reçu un message que l'on aurait dû recevoir)

Byzantins

On suppose des rondes et de la diffusion



Accord byzantin

Algorithme de décision

Accord: les processus corrects décident la même valeur

Validité : Si tous les processus corrects proposent la même valeur v alors la seule décision possible est v

Terminaison : tous les processus corrects décident

Général byzantin

Une autre version… (équivalente: exercice)

Un général donne un ordre à ses lieutenants.

(Loyal-honnête-correct traître sinon)

Algorithme tel que:

Accord: les lieutenants loyaux décident de la même valeur

Validité : Si le général est loyal (n'est pas un traitre) la seule décision possible des lieutenants loyaux est la valeur du général

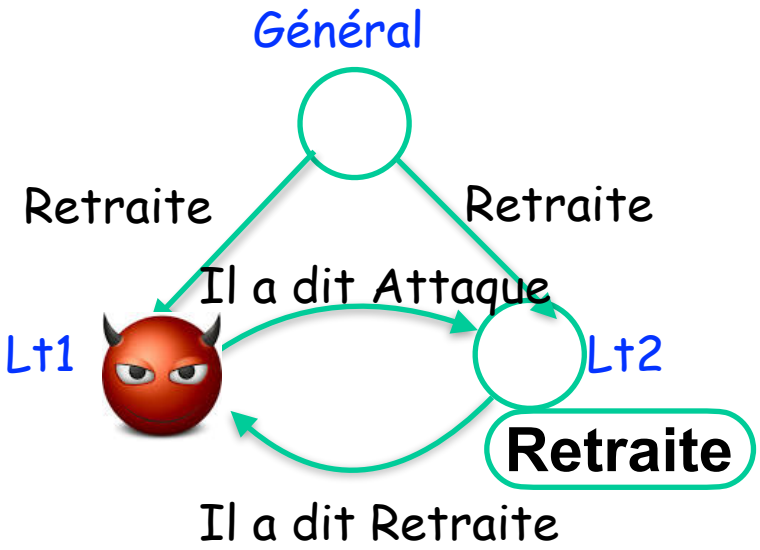
Terminaison : tous les lieutenants loyaux décident

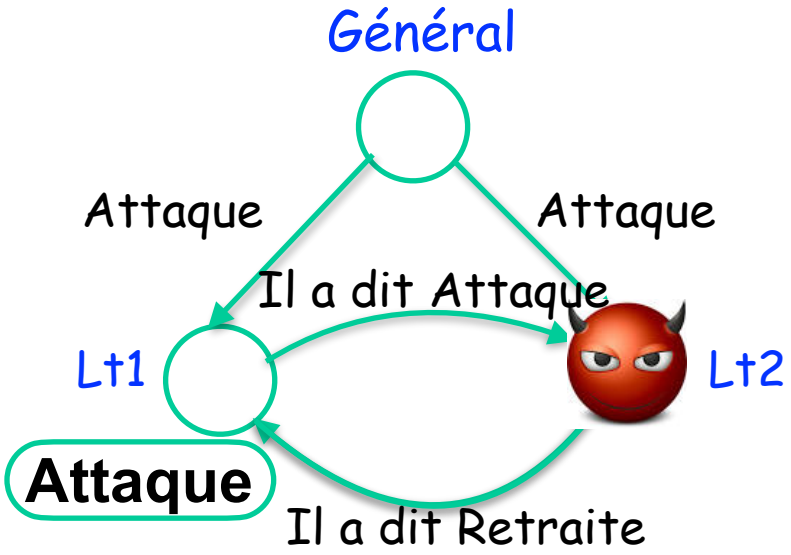
Un général + $n-1$ lieutenants au plus t traîtres (attention le général peut être un traître)

Essayons…

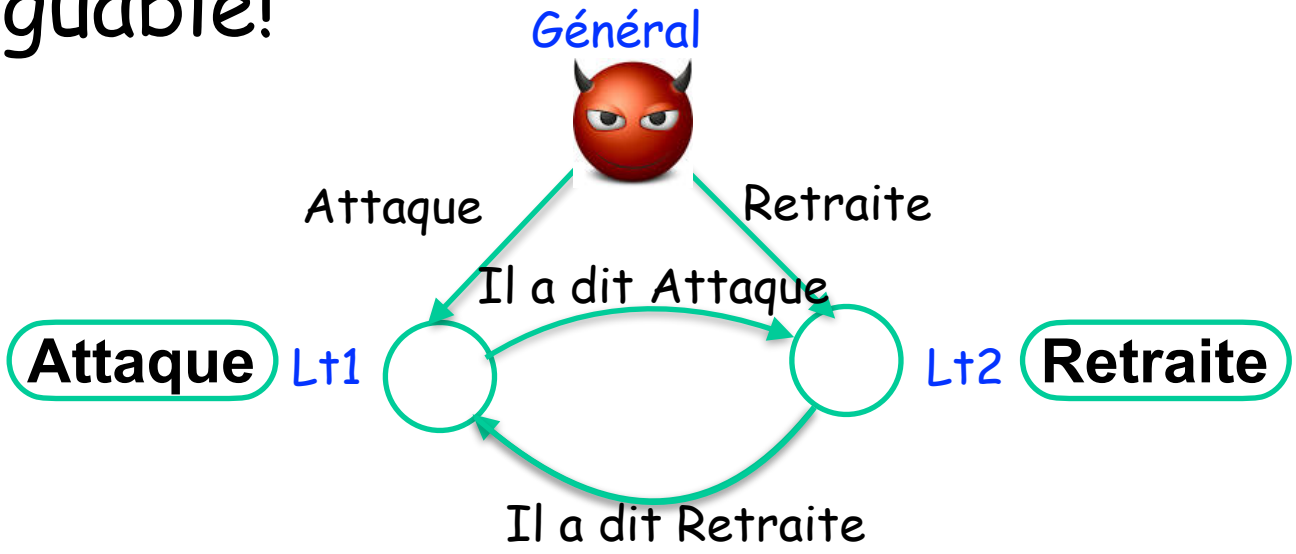
Pour $n = 2$ et $t = 1$, c'est clairement impossible. (Pourquoi?)

Essayons pour $n = 3$ et $t = 1$…





Indistinguable!

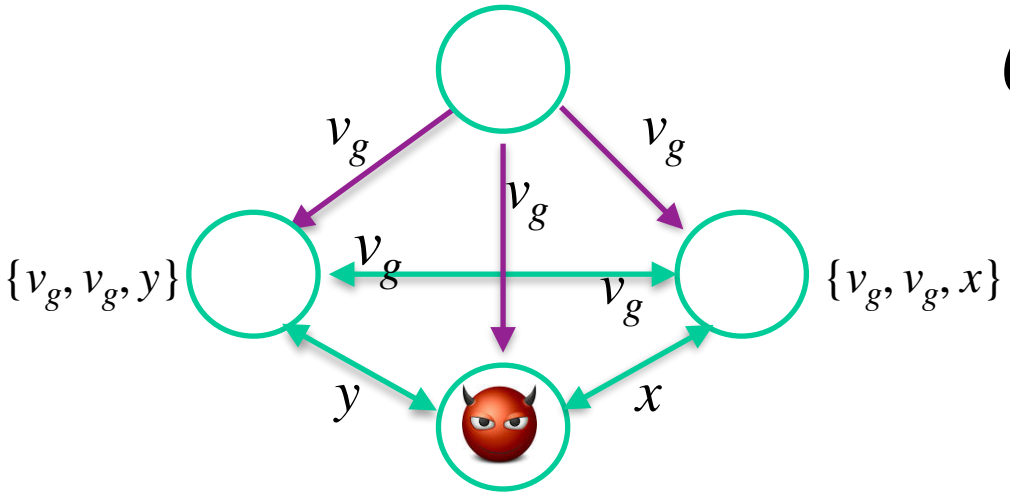


Résultat d'impossibilité

En généralisant… on obtient:

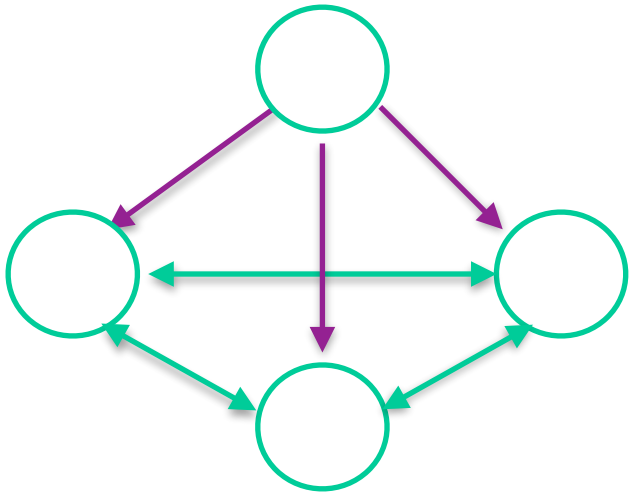
Théorème: si $n \leq 3t$ il n'y a pas d'algorithme pour le problème du général byzantin

(t=1, n=4)

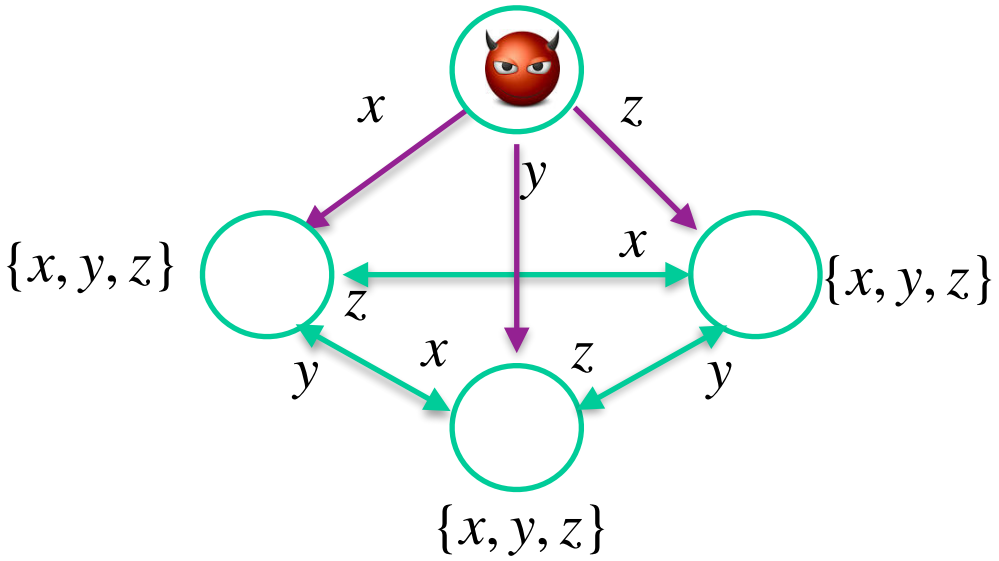


Général honnête:

- Tous les lieutenants ont v_g
- Chaque lieutenant a au moins une majorité v_p



(t=1, n=4)



Général traître:

- Les lieutenant sont honnêtes!
- Ils obtiennent dans la deuxième ronde le même ensemble de valeurs -> accord

Si une valeur majoritaire:
choisir cette valeur
Sinon valeur par défaut

Et pour les autres valeurs de n et t?

On a une solution pour t=1 et n (n≥4) comment faire pour t et n>3t?

En généralisant on obtient par récurrence:

Algorithme A(t) (t nombre de panne)

A(0)

- Le général envoie sa valeur aux lieutenants
- Les lieutenants prennent cette valeur

A(t)

- Le général envoie sa valeur v_g aux lieutenants
- Soit v_i la valeur reçu du général par le lieutenant L_i. L_i agit avec v_i comme général pour A(t-1) avec les $(n-2)$ autres lieutenant
- Soit v_j la valeur obtenue par L_i de L_j pour A(t-1) avec L_j comme général: L_i choisit la valeur majoritaire parmi les v_j

Réplication…

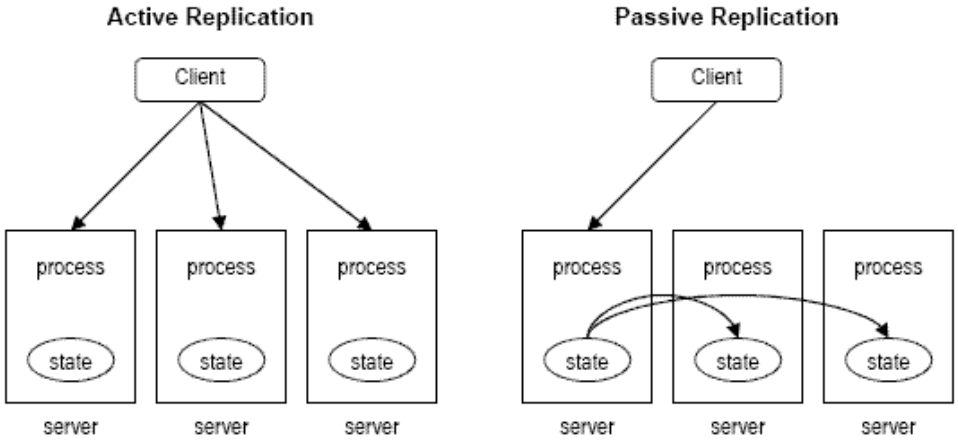
Si certains processus peuvent tomber en panne (crash failure).

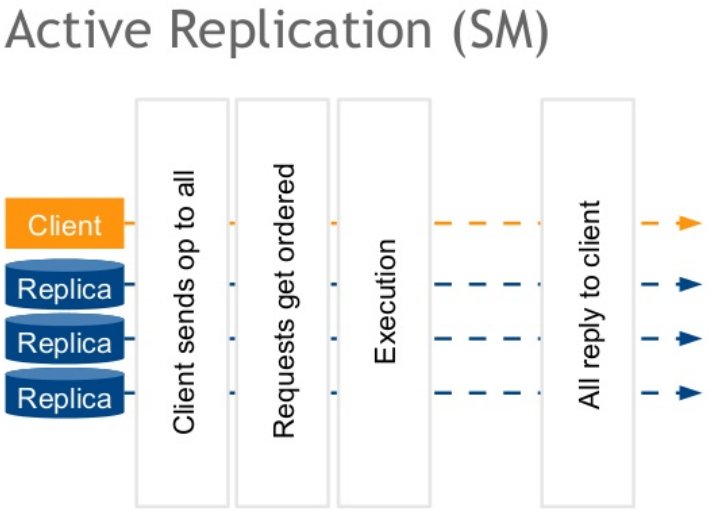
Assurer qu'un service est assuré malgré les pannes.

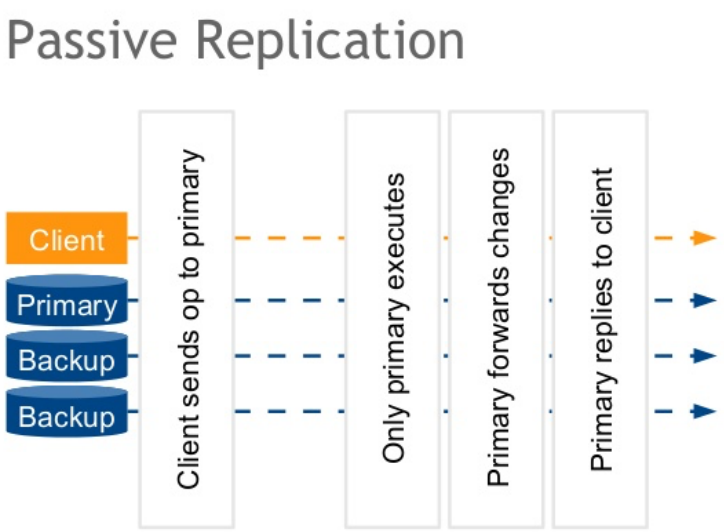
- Réplication: le serveurs sont répliqués de façon à ce que chaque client reçoit une réponse du service de façon cohérente

Tolérance aux défaillances..

Réplication active et passive



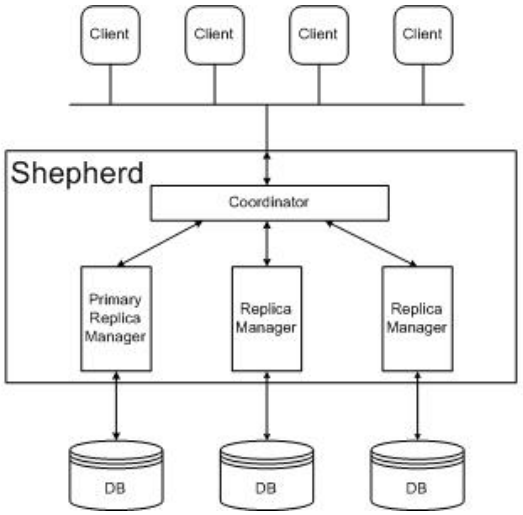




réplication passive

« Primary-Backup » (très simplifié)

- Un seul serveur est actif (primary ou maître)
- Les autres sont passifs (backup ou esclaves) et enregistrent passivement l'état du « primary »
- Si un primary tombe en panne…
  - Choisir un nouveau « primary » parmi les backups
  - Démarrer le primary à partir de l'état de backup
- Problème: Assurer l'atomicité des requêtes (Que se passe-t-il si le primary tombe en panne pendant une requête?)



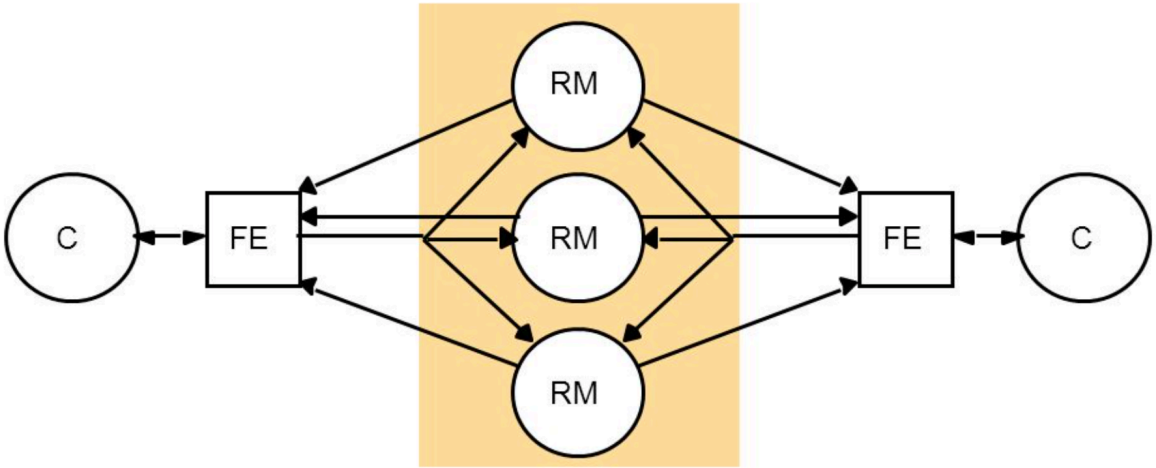
réplication Active

Tous les server sont actifs et traitent toutes les requêtes dans le même ordre

- Tous les serveurs passent par les mêmes états (ils ont la même histoire)

Diffusion atomique:

- Chaque requête arrive dans le même ordre sur chaque processus (serveur) : (1) tous les processus serveurs ont la même histoire (2) les réponses aux requêtes sont les mêmes



Diffusion fiable (Reliable Broadcast)

primitives Rbcast Rdeliver:

Accord: Si p correct Rdeliver m alors tout processus correct Rdeliver m

- *Validité:* Si p correct Rbcast m alors p Rdeliver m
- *Intégrité*: Si p Rdeliver m alors un processus q a fait un Rbcast m

RBcast

Algorithm for process p:

To execute Rbcast(m)
    send (m) to p

Rdeliver(m) occurs when
    upon receive(m) **do**
        if has not previously executed Rdeliver(m)
        then
            send (m) to all
            Rdeliver(m)

Diffusion atomique

(Atimic broadcast) primitives ABcast ABdeliver:

- RBcast est une diffusion fiable telle que :
- *ordre total:* Si p et q ABdeliver m et m' alors si p ABdelivers m avant m' alors q ABdelivers m avant m'

ABCast est « universel »: il permet de réaliser la duplication active: simuler un serveur séquentiel sans pannes à partir de serveurs dupliqués (avec pannes)

- state machine replication:
  - Une « state machine » séquentielle A
  - t+1 processus simulent A
  - Chaque requête est faite par diffusion atomique
  - On obtient une « implémentation t-résiliente de A

Diffusion atomique et Consensus

Rappel: consensus est un algorithme de décision

Terminaison: Tous les processus corrects décident

Validité: Si p décide v alors v est une valeur initiale d'un processus

Accord: Si p et q decident, ils décident la même valeur

Diffusion atomique et consensus

Diffusion atomique et consensus sont équivalents:

- De la diffusion atomique au consensus (exercice)
- Du consensus à la diffusion atomique

Diffusion atomique à partir de consensus et de diffusion fiable

Algorithm for process p:

Initialization:
$RDelivered := \emptyset$
$ADeliveres := \emptyset$

To execute Abcast(m)
Rbcast(m)

Adeliver(_) occurs when
upon Rdeliver(m) **do**
$RDelivered := RDelivered \cup \{m\}$

do forever
$AUndelivered := RDelivered - ADelivered$
if $AUndelivered \neq \emptyset$ **then**
$k := k + 1$
$propose(k, AUndelivered)$
wait for $decide(k, msgSet)$
$batch(k) := msgSet - ADelivered$
A-deliver all messages in $batch(k)$ in some deterministic order
$ADelivered := ADelivered \cup batch(k)$

Consensus?

On a vu que dans le cas synchrone on peut faire du consensus en présence de pannes crash (floodset)

Dans le cas asynchrone:

- consensus s'il n'y a pas de défaillances?
  - Avec des défaillances ?

Mais…

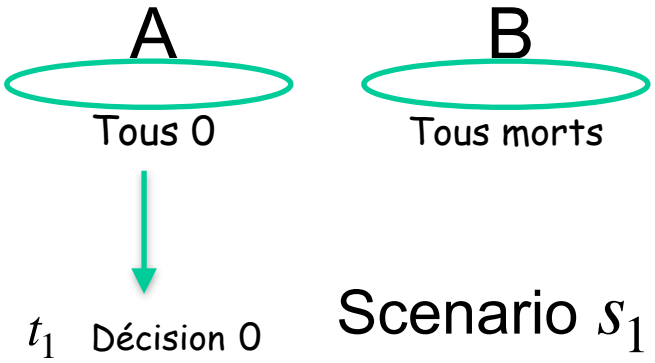
Théorème: (Fischer, Lynch, Paterson)

il n'existe pas d'algorithme de consensus tolérant même à une panne dans un système asynchrone.

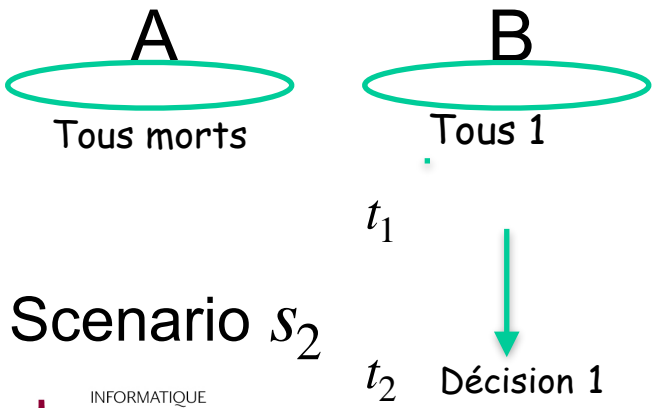
Avec 100000 processus et un seul processus peut tomber en panne il est impossible de réaliser le consensus même sur 0 ou 1.

Un cas facile:

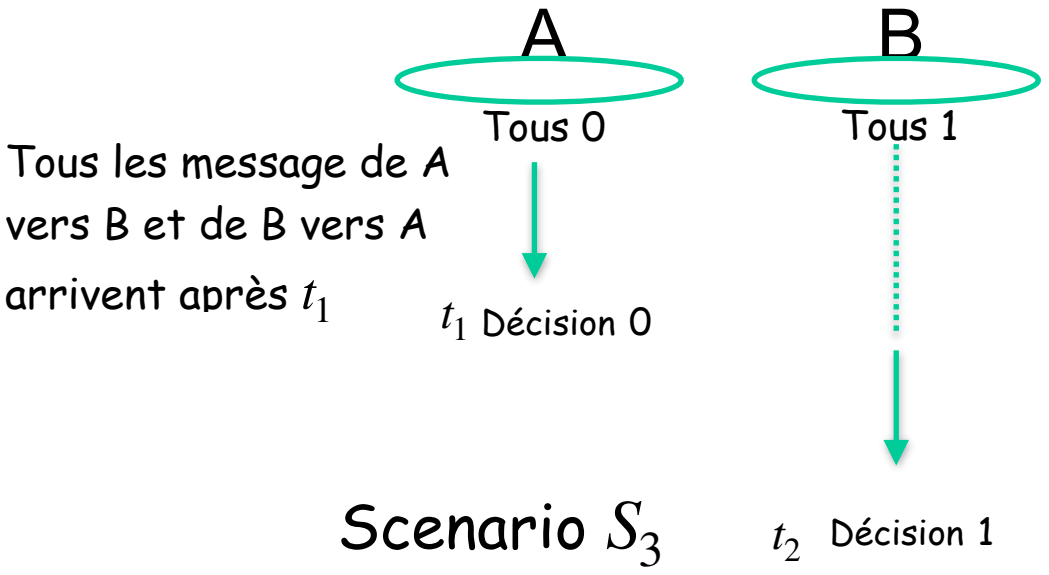
Si le nombre de panne t est tel que $t > n/2$ l'ensemble des processus peut se décomposer en deux sous-ensemble A et B tels que tous les éléments de A ou tous les éléments de B sont en pannes.



Scenario s_1



Scenario s_2



Scenario S_3

Conséquences

Dans un système asynchrone même avec au plus une panne d'un processus (1-résilient)

- On ne peut pas faire de consensus.
- On ne peut pas faire de diffusion atomique
- On ne peut pas faire de réplication active

Conséquences (suite)

Pour réaliser du consensus (avec des pannes crashs)

- Système synchrone
- Sytème partiellement synchrone
  - Exemple: un jour le système est « synchrone »
- Détecteur de défaillances
- Consensus probabiliste (terminaison avec probabilité 1)

Remarques:

On a supposé que la communication se faisait par envoi-réception de messages. Qu'en est-il si la communication se fait par partage d'objets (shared memory) comme dans le cours de programmation répartie?

Message-passing shared memory

Les modèles avec envoi-réception de messages (message-passing) et les modèles avec mémoire partagée (shared memory) sont « équivalents » (avec une majorité de processus corrects):

On peut « simuler » des registres atomiques en massage-passing et une majorité de processus corrects

Simuler un registre 1-reader 1-writer

For the writer
    to write(v)
        $seq := seq + 1$
        send (W, v, seq) to all
        wait until receiving $\lfloor n/2 \rfloor + 1$ messages (ACK, seq)

For the reader
    to read()
        send(R) to all
        wait until receiving $\lfloor n/2 \rfloor + 1$ messages (V, v, s) such that $s > seq$
        return val
            such that (V, val, S) has been received
            and S is the max of the sequence number of received V messages

For all processes
    when (W, v, s) is received
    if $s > seq$ then
        $val := v; seq := s$
        send (ACK, s)

    when(R) is received
        send (V, val, seq) to p_r

Simuler un registre…

Un résultat classique montre qu'on peut implémenter un registre atomique multi-writer multi-reader à partir de registres atomiques 1-writer 1-reader.

Avec une majorité de processus corrects message-passing et shared -memory sont équivalents.

Impossibilité du consensus dans les deux modèles pour $t > 1$ (au plus une panne crash)

A majority of correct processes is needed

partition argument:

- if $n \leq 2t$ then we can partition the set of processes in two set S_1 and S_2 such that $|S_1| \geq t$ and $|S_2| \geq t$.
  - Run A_1: all processes in S_1 are correct and all processes in S_2 are initially dead, p_0 invokes a write(1), at some time t_0 the write terminates
  - Run A_2: all processes in S_2 are correct and all processes in S_1 are initially dead, p_1 S_2 invokes a read(), at time t_0 +1 the read terminates at time t_1
  - Run B: « merge » of A_1 and A_2 but no process crash. write(1) terminates before the read() and the read return 0

contradiction

Complément

Algorithme de Ben Or: consensus avec terminaison presque sûre.

Ben Or

Every process p executes the following:

0 **procedure** *consensus*(v_p) {v_p is the initial value of process p}

1 $x \leftarrow v_p$ {x is p's current estimate of the decision value}

2 $k \leftarrow 0$

3 **while** true **do**

4 $k \leftarrow k + 1$ {k is the current phase number}

5 **send** (R, k, x) **to** all processes

6 **wait for** messages of the form $(R, k, *)$ **from** $n - f$ processes {"$*$" can be 0 or 1}

7 **if** received more than $n/2$ (R, k, v) with the same v

8 **then send** (P, k, v) **to** all processes

9 **else send** $(P, k, ?)$ **to** all processes

10 **wait for** messages of the form $(P, k, *)$ from $n - f$ processes {"$*$" can be 0, 1 or ?}

11 **if** received at least $f + 1$ (P, k, v) with the same $v \neq ?$ **then** *decide*(v)

12 **if** at least one (P, k, v) with $v \neq ?$ **then** $x \leftarrow v$ **else** $x \leftarrow 0$ or 1 randomly {query r.n.g.}