Chapitre 6

(très) rapide introduction à la tolérance aux fautes

Tolérance aux fautes

Les réseaux sont naturellement sujets à des dysfonctionnement

- Les réseaux modernes sont à grande échelle, fait de machines hétérogènes et produites en masse à faible coût
- Internet : 350 millions de serveurs en 2006
  12,3 milliards d'objets connectés fin 2021
- Réseaux sans fils :
  Communication radio : beaucoup de pertes de messages
  Crash de machines à cause des batteries limitées

➔ Forte probabilité de pannes
➔ Intervention humaine impossible

Tolérance aux fautes – Dysfonctionnement ?

Une faute provoque une erreur qui entraîne une défaillance

Défaillance d'un composant = son comportement n'est plus conforme à sa spécification

- Composant = lien de communication ou nœud (processus, machine)
  - Un **processus** arrête d'exécuter un programme.
  - Un **lien de communication** perd un message.
- A l'opposé : nœud correct, lien fiable

Erreur = état du système à partir duquel la poursuite de l'exécution est susceptible de conduire à une défaillance.

- chaînage d'une liste corrompu, pointeur non initialisé : *erreurs logicielles*
- câble réseau déconnecté, unité disque éteinte : *erreurs matérielles*

Faute = événement ayant entrainé une erreur

- faute de programmation (pour les erreurs logicielles)
- événements physiques, e.g., usure, malveillance, catastrophe (erreurs matérielles)
- Ex: coupure d'un lien, perturbation électro-magnétique …

Tolérance aux fautes – Fautes

Origine de la faute = type de composant responsable de la faute
→ lien de communication ou processus

Cause de la faute =

- **bénigne** : non volontaire, (ex: problème matériel)
- **maligne** : due à une intention (malveillante ou malicieuse) extérieure au système

Ex: **fautes byzantines** = dues à des processus « byzantins »
Qui ont un comportement arbitraire,
Ne suivant plus le code de leurs algorithmes locaux
Peut venir d'une erreur matérielle, un virus, la corruption du code…

Tolérance aux fautes – Fautes

Durée de la faute =

→ Faute définitive / franche : durée de faute >= temps restant d'exécution du protocole. Se produit de manière isolée (~ une fois pendant le temps d'exécution du protocole)

Ex: Le processus cesse définitivement de faire des calculs (crash)

→ Faute transitoire / intermittente : sinon. (Se produit plusieurs fois pendant le temps d'exécution du protocole)

Ex: Perte intermittente de messages : régulièrement un lien perd des messages

Ex pour un processus : comportement erroné de composants du réseau. Durant une certaine période (finie). Après cette période, les composants reprennent un comportement correct. Cependant, l'état du système en est altéré.

Détectabilité = une faute est détectable si son incidence sur la cohérence de l'état d'un processus permet à celui-ci de s'en apercevoir.

Solution tolérante aux fautes

Prise en compte **automatique** de la possibilité de pannes au niveau algorithmique (éviter de réinitialiser le réseau après chaque panne)

→ Concevoir des algorithmes capables d'assurer un service malgré les pannes

- A priori, un système n'est généralement que partiellement touché. Les sites corrects peuvent continuer à fonctionner.

  Et ils peuvent prendre en charge les tâches allouées aux sites défaillants…
- → perte de performance mais pas (nécessairement) fonctionnement erroné

Approches :

- Robuste : masquer l'effet des pannes, la spécification est toujours garantie
- Autostabilisante : la panne provoque un dysfonctionnement temporaire, mais le système revient à son comportement attendu en temps fini

Fautes : Modèles de fautes pour les **processus**

Crash failure :

= arrêt prématuré de l'exécution ; avant l'exécution le processus s'exécute normalement

- *crash stop* : le processus s'arrête définitivement
- *noisy crash* : quand le processus s'arrête il a le temps de prévenir ses voisins
- *crash recovery* : le processus peut recommencer à s'exécuter

  → perte de l'état interne possible

  → pour éviter que le processus ne soit réinitialisé et n'envoie des messages contradictoires avec ce qui a précédé, on suppose souvent qu'il a en plus un *espace mémoire stable* qui ne sera pas affecté par le crash (mais qu'il faudra maintenir tout au long de l'exécution)

Arbitrary failure : (byzantine / malicious failure) *malicieuse quand la faute est intentionnelle*

- le processus ne s'exécute pas selon son algorithme

Fautes : Modèles de fautes pour les **les canaux de communication**

Grâce à la détection d'erreurs et aux codes correcteurs, les messages corrompus peuvent être corrigés et reçus correctement. Si un message ne peut pas être corrigé, il peut être jeté et apparaître comme un message perdu.

- Ainsi les fautes dans les canaux sont ramenées à des pertes de messages et problèmes similaires = duplication, création
- **Perfect channels** : pas de perte, pas de création, pas de duplication, pas de modification
- **Fair lossy channels** : infinite send ⇒ infinite receive ; finite duplication ; no creation

Comment obtenir un lien parfait en supposant Fair Lossy Channel ?

1ère experience : décision commune malgré perte de message

Peut on établir un protocole qui termine ??

2 processus, p1 ————> p2

p1 peut envoyer des messages à p2 mais le canal peut perdre des msgs.

p1 : send(m) ;

p2 : receive(m) ;

Sans perte de message, ce simple code suffit pour que la donnée contenue dans m soit partagée entre les 2 processus - le protocole d'échange est bien évidemment fini !

Avec perte de message, on veut les mêmes objectifs :

p1 envoie m. ……. protocole d'échange …… p2 a reçu m ……

le protocole d'échange s'arrête pour p1 et pour p2

Décision commune malgré perte de message un protocole qui termine ??

Comment faire ?

- p2 doit pouvoir communiquer vers p1 pour lui dire qu'il a reçu, sinon p1 ne pourra jamais s'arrêter

Essai1 de Protocole d'échange pour p1

repeat send(m) to p2 until receive(ack) from p2 end repeat;

→ si tant est que le ack de p2 est reçu, cet algorithme s'arrête

Essai1 de Protocole d'échange pour p2

receive(m) from p1;

send(ack) to p1;

→ Ok ?

Décision commune malgré perte de message un protocole qui termine ??

Comment faire ?

- p2 doit pouvoir communiquer vers p1 pour lui dire qu'il a reçu, sinon p1 ne pourra jamais s'arrêter

Essai1 de Protocole d'échange pour p1

repeat send(m) to p2 until receive(ack) from p2 end repeat;

→ si tant est que le ack de p2 est reçu, cet algorithme s'arrête

Essai1 de Protocole d'échange pour p2

receive(m) from p1;

repeat send(ack) to p1 until ??? end repeat;

??? should be receive(ack2) from p1

⇒ algo of p1 should be modified to send ack2:

ack2 will be sent until some new ack, say ack3 is received from p2, etc

⇒ BAD solution!

Décision commune malgré perte de message un protocole qui termine ??

Comment faire ?

- p2 doit pouvoir communiquer vers p1 pour lui dire qu'il a reçu, sinon p1 ne pourra jamais s'arrêter

Essai1 de Protocole d'échange pour p1

repeat send(m) to p2 until receive(ack) from p2 end repeat;

Essai2 de Protocole d'échange pour p2

receive(m) from p1

repeat send(ack) to p1 each time (m) is received from p1; end repeat;

This solution solves the problem of multiple types of messages

But still, none of the processes stops.

⇒ BAD solution!

Décision commune malgré perte de message un protocole qui termine ??

Impossibility results:

*There is no **terminating** algorithm that solves our problem using channels prone to arbitrary message losses.*

Proof intuition: By contradiction.

Assume there exists such algo. Let us consider an execution. As the algo terminates, the execution contains a last message exchange : call it **m**, let say **from p1 to p2** (the arguments are the same if it were from p2 to p1).

p1 sends m - executes some internal computations maybe - then terminates

→ the fact that m has been received by p2 does not affect p1's termination, so p1 could have decided to stop before sending m

p2 receives m - executes some internal computations maybe - then terminates

→ the fact that p2 receives m is not required for the termination of the algorithm (in particular for p1 which does not wait any ack since p2 terminates right after). Hence p2 could have stopped before receiving m.

⇒ m is no longer useful to detect termination and can be removed from the execution. Iteratively doing so on the execution produces an execution with no message exchange at all which is still correct from the termination point of view but lacks the fact that both processes should have access to the data in m.

Protocole d'échange pour les canaux fair lossy: Création d'un lien parfait

Ce qu'on vient de toucher du doigt:

→ dès qu'un canal perd des messages de façon arbitraire, on ne peut pas avoir un protocole d'échange qui termine !!

➔ Construction d'un protocole d'échange

- Protocole point à point, càd pour un lien, dans une direction
- A partir d'un canal fair lossy
- On crée un protocole d'échange parfait
- (mais qui ne termine pas)

Fair Lossy Channel Specification

Rappel : les processus peuvent être incorrects (crash-stop)

Primitives : send, receive

- **Fair-loss:** si un processus correct p1 envoie (send) un message m infiniment souvent à un processus correct p2, alors p2 délivre (receive) m un nombre infini de fois

⇒ un canal ne perd pas systématiquement un message : si l'envoyeur et le récepteur sont tous les 2 corrects, quand l'envoyeur répète continument son envoi, le message finira par être reçu

- **Finite duplication:** si un processus correct p1 envoie (send) un message m un nombre fini de fois à un process p2, alors m ne peut pas être délivré (receive) à p2 un nombre infini de fois

⇒ intuitivement, le canal ne duplique pas plus le message que la duplication due à la retransmission

- **No creation:** si un processus p2 délivre (receive) un message m qui a été envoyé par un processus p1, alors m a été envoyé (send) préalablement à p2 par p1

⇒ pas de message créé ou corrompu par le canal

Perfect Channel Specification

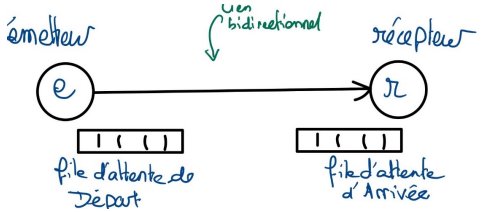
Rappel : les processus peuvent être incorrects (crash-stop)

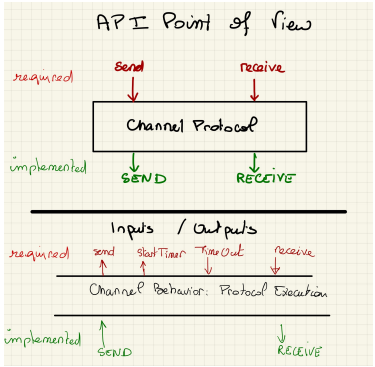
Primitives : SEND, RECEIVE

1. **Reliable delivery:** si un processus correct envoie (SEND) un message m à un processus correct p2, alors p2 finit par délivrer m (RECEIVE)
2. **No duplication:** aucun message n'est délivré (RECEIVE) par un processus plus d'une fois
3. **No creation:** si un processus p2 délivre (RECEIVE) un message m venant d'un envoyeur p1, alors m a été préalablement envoyé à p2 par p1

Algorithme : un canal parfait à partir d'un canal fair lossy

- Utilise : send, receive (fair lossy)
- Implémente: SEND, RECEIVE (perfect)





Algorithme : un canal parfait à partir d'un canal fair lossy

Init:
sent := empySet;
delivered := emptySet;
3 programmes en //
- **Repeat (timer)**
  forall (q, m) in sent do:
  send(m) to q;
- **SEND(m) to q:**
  send(m) to q;
  sent := sent U { (q, m) };
- **Forever:**
  receive(m) from q;
  if m not in delivered then
  delivered := delivered U { m };
  RECEIVE(m) from q;

Algorithme : un canal parfait à partir d'un canal fair lossy

Init:
sent := empySet;
delivered := emptySet;
3 programmes en //
- **Repeat (timer)**
  forall (q, m) in sent do:
  send(m) to q;
- **SEND(m) to q:**
  send(m) to q;
  sent := sent U { (q, m) };
- **Forever:**
  receive(m) from q;
  if m not in delivered then
  delivered := delivered U { m };
  RECEIVE(m) from q;

Proof (sketch): Reliable delivery
si un processus correct envoie (SEND) un message m à un processus correct p2, alors p2 finit par délivrer m (RECEIVE)

Fair-lossy (rappel) : si un processus correct p1 envoie (send) un message m infiniment souvent à un processus correct p2, alors p2 délivre (receive) m un nombre infini de fois

⇒ quand SEND(m) est exécuté, le msg est inséré dans l'ensemble sent.
Ensuite et pour toujours, régulièrement, ce msg est renvoyé.

⇒ si on considère un processus correct, alors il va envoyer (send) le message un nombre infini de fois et par propriété fair-lossy, si le processus au bout du lien est lui aussi correct, celui-ci va le recevoir (receive). Quand un message est reçu (receive), il est délivré (RECEIVE) si cela n'a pas déjà été fait. D'où la propriété de reliable delivery.

Algorithme : un canal parfait à partir d'un canal fair lossy

Init:
sent := empySet;
delivered := emptySet;
3 programmes en //
- **Repeat (timer)**
  forall (q, m) in sent do:
  send(m) to q;
- **SEND(m) to q:**
  send(m) to q;
  sent := sent U { (q, m) };
- **Forever:**
  receive(m) from q;
  if m not in delivered then
  delivered := delivered U { m };
  RECEIVE(m) from q;

Proof (sketch): No duplication
aucun message est délivré (RECEIVE) par un processus plus d'une fois

→ assuré par le mécanisme de la variable "delivered"

Proof (sketch): No creation
si un processus p2 délivre (RECEIVE) un message m venant d'un envoyeur p1, alors m a été préalablement envoyé à p2 par p1

→ Trivial (c'est la même propriété pour les 2 spécifications)

Algorithme : un canal parfait à partir d'un canal fair lossy

On vient de montrer qu'on pouvait construire un canal parfait à partir d'un canal fair-lossy.

Nb: On peut faire plus efficace et éviter que des messages reçus soient renvoyés à l'infini, cf protocole du bit alterné

Exemple : Bit alterné

Perte intermittente de messages, non détectable a priori,
Approche robuste (masquante)

- Perte **équitable** de messages = si des messages sont envoyés infiniment souvent, alors une infinité de messages est livrée

- A partir des primitives send/receive non robustes, on fabrique des primitives send/receive qui assurent que le lien est **fiable** =

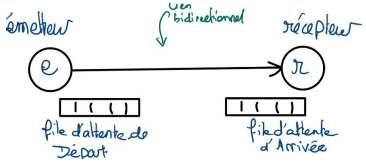
Pas de création : Tout message reçu par un processus p venant du processus q a été envoyé au préalable par q à p.

Pas de duplication : Tout message est reçu au plus une fois.

Pas de perte : Tout message envoyé est livré au récepteur en temps fini

Exemple : Bit alterné

Chaque noeud est muni d'une variable booléenne (x pour e, y pour r)



E envoie la donnée d à r :

x := !x

répéter : send<DATA, d, x> to r jusqu'à recevoir <ACK, x> de r

la repetition se fait régulièrement (timer)

R reçoit un message <DATA, d, x> de r :

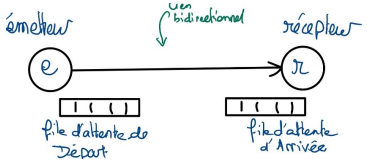
si x <> y */* 1ère fois que r reçoit ce message */* alors

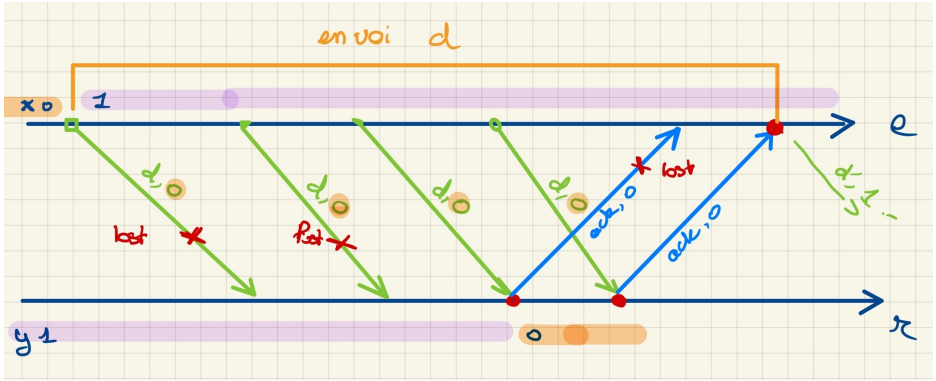
d est inséré dans la file; y := x; send<ACK, x> to e

si x == y */* d a déjà été insérée mais le ack non reçu */* alors

send<ACK, x> to e

Exemple : Bit alterné





Algorithme : un canal parfait à partir d'un canal fair lossy

On a montré qu'on pouvait construire un canal parfait à partir d'un canal fair-lossy.

- De façon très inefficace
  L'efficacité souvent n'est pas le sujet → on montre que c'est possible
  Démarche classique : et ensuite si besoin on améliore les performances !

- *Un peu plus efficace :* protocole du bit alterne
  (mais bien sûr qui ne termine pas)

⇒ A partir de là, on peut supposer que ***les liens sont parfaits entre 2 processus corrects***

→ Les solutions proposées pourront fonctionner avec des canaux fair lossy, en utilisant un transformateur (fair lossy → perfect)