Huitième partie

Systèmes répartis et tolérance aux fautes

Contenu de cette partie

On building systems that will fail... *(F. Corbató, prix Turing 1990)*

- Vocabulaire
- Principes de mise en œuvre de la tolérance aux fautes
- Reprise après panne
- Serveurs à haute disponibilté

Sources, références, compléments

- *A. S. Tanenbaum & M. van Steen,*
  Distributed Systems - Principles and Paradigms,
  ch. 8 : Fault Tolerance,
  accès libre : https ://www.distributed-systems.net
- *S. Krakowiak / A. Kshemkalyani & M. Singhal : cf début du cours*

Plan

Modèles de pannes

Pannes arbitraires (ou *byzantines*)

Une partie du système peut faire n'importe quoi
(y compris avoir un comportement malveillant)

- représente les conditions les plus défavorables (*pire cas*)
  → modèle utilisé pour des systèmes critiques (nucléaire, spatial...)
- nécessite une redondance élevée
  rappel : il faut 3f+1 processus pour résister à f pannes byzantines en synchrone

Pannes de temporisation

L'écart par rapport aux spécifications concerne uniquement le temps.

temps de réaction à un événement, respect d'échéances...

Panne d'omission

Le système perd des messages entrants et/ou sortants

Modèle considéré

Panne franche (ou *panne d'arrêt (fail stop)*)

Soit le système comportement correct, n'omet pas de message etc. . . soit il est en panne (défaillant), et ne fait rien

Remarques

- Selon les hypothèses quant au redémarrage, on peut distinguer
  - la panne avec amnésie : le système repart toujours du même état initial, indépendant de l'état au moment de la panne
  - la panne avec reprise de l'état au moment de la panne
  - la panne définitive, sans reprise
- la panne franche est le cas le plus simple à détecter ($\rightarrow$ à traiter)
  - $\rightarrow$ technique *fail-fast* classique : forcer l'arrêt dès qu'une erreur interne est détectée

Analyse des pannes : définitions

Erreur

(partie de l') état du système susceptible de provoquer une panne

$\rightarrow$ état dont les propriétés ne sont pas conformes aux spécifications

Exemples

- logiciel : indice calculé supérieur à la dimension d'un tableau
- matériel : connexion coupée

Faute

Toute cause (événement, circonstance) pouvant provoquer une erreur

Exemples

faute de programmation, malveillance, accident etc.

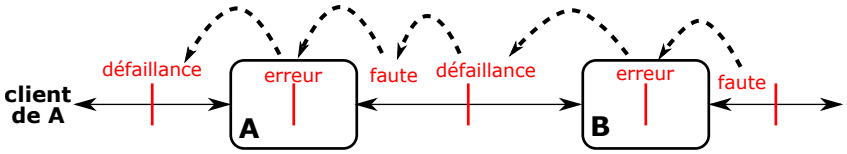
De l'erreur à la défaillance

Une erreur n'entraîne pas nécessairement une panne :

- un état erroné peut ne pas être atteint par une exécution.
- il peut y avoir un délai (*de latence*), entre l'apparition de l'état d'erreur et la panne (on dit alors que l'erreur est *latente*)

Propagation (cascade) de fautes

- le bon fonctionnement de A dépend de celui de B
- la défaillance de B (service incorrect) constitue une faute pour A
- qui peut à son tour provoquer une erreur interne à A, puis une défaillance de A



faute $\rightarrow$ *erreur* $\rightarrow$ *défaillance* $\rightarrow$ *faute* $\rightarrow$ *erreur* $\rightarrow$ . . .

Comment garantir qu'un système reste fiable, sûr et disponible ?

Évitement des fautes : essayer d'empêcher les fautes de se produire

- Analyser les causes potentielles de fautes
- Prendre des mesures pour les éliminer ou réduire leur probabilité

Tolérance aux fautes : assurer le service malgré l'existence de fautes

Moyen : redondance

- redondance d'information (détection d'erreur)
- redondance temporelle (traitements multiples)
- redondance matérielle (composants dupliqués)

La tolérance aux fautes est indispensable à la sûreté de fonctionnement

- L'*évitement* des fautes est *coûteux*
- L'*occurrence* de fautes est *inévitable*

$\rightarrow$ concevoir les systèmes pour qu'ils fonctionnent (éventuellement de manière dégradée), même en présence de fautes

Mise en œuvre de la tolérance aux fautes

Techniques de base

Récupération (*error recovery*)

- *Détecter* l'erreur
  - comparaison des résultats de composants dupliqués
  - test de vraisemblance
    - explicite (exprimer et vérifier des propriétés d'état spécifiques)
    - implicite (anomalies observées sur échéances, accès mémoire...)
- *Remplacer* l'état d'erreur par un état correct
  - à partir d'un état précédent enregistré : *reprise*
  - en reconstruisant un état courant correct : *poursuite*

Compensation (*error masking*)

Le système a une redondance interne suffisante pour corriger l'erreur de manière transparente pour les utilisateurs

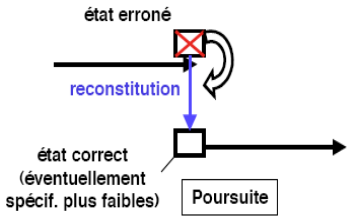
- composants dupliqués (→ réduire la probabilité d'effet d'une faute)
- traitements dupliqués (même opération avec algos/matériels différents)

Récupération

Poursuite (*forward recovery*)

- (tentative de) *reconstitution* d'un état correct, à partir de l'état courant
- la reconstitution est souvent partielle → service dégradé
- ad-hoc : reconstitution spécifique à chaque application

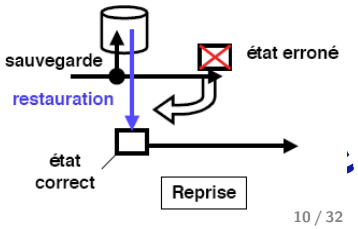
Exemple (perte de messages dans un flux)
Le récepteur interpole les paquets manquants à partir de leurs voisins.



Reprise (*backward recovery*)

- retour en arrière vers un état antérieur dont on sait qu'il est correct
- nécessite la sauvegarde d'états corrects
- technique générale

Exemple (perte de messages dans un flux)
L'émetteur stocke les messages émis pour pouvoir les réémettre sur demande du récepteur



Mise en œuvre de la reprise

Reprise

- le système est ramené à un état précédant l'occurrence de l'erreur (*retour arrière*)
- cet état doit avoir préalablement été sauvegardé (*points de reprise*)

Difficultés

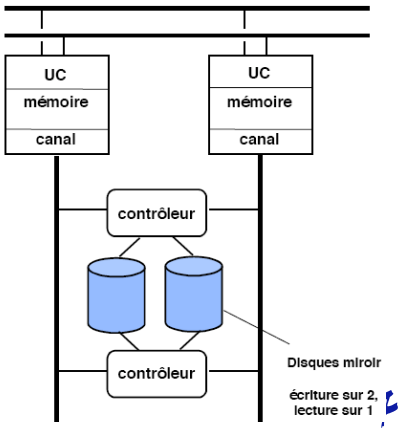
- sauvegarde et restauration doivent être *atomiques*
- la *construction* des points de reprise doit elle-même être *protégée contre les fautes*
- dans un système réparti
  - chaque site gère des clichés locaux
  - l'ensemble de ces points de reprise locaux doit permettre de construire un état global cohérent
    → *garantir/évaluer la coordination de la prise des clichés locaux*

Exemple de récupération par reprise : *Tandem Non-Stop*

Tolère la panne d'un composant (UC, bus, mémoire, contrôleur, disque)

Matériel

- alimentation électrique redondante aussi (deux sources indépendantes)
- composants à détection d'erreur par contrôle de vraisemblance
- une détection d'erreur bloque immédiatement le composant (*fail fast*)



Logiciel

- Système d'exploitation spécifique : Unix gérant les processus par paires (processus actif - processus de secours)

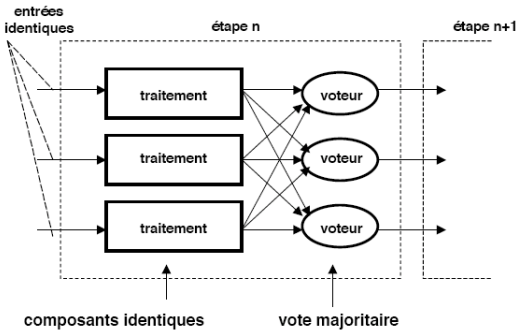
Compensation

traitements redondants, pour aboutir à au moins un traitement correct

- \+ transparent, sans délai
- − coût de la duplication

Exemple : architecture TMR (Triple Modular Redundancy)

- vote majoritaire
- résiste à la défaillance d'un composant de traitement ou d'un voteur sur l'ensemble des étapes



Plan

Récupération par reprise après panne dans un système réparti

Principe

- sauvegarde d'un état global
- panne → restauration, puis reprise à partir de l'état sauvegardé

Contraintes

- L'état sauvegardé doit être cohérent
- Le coût de la sauvegarde doit être réduit
  → limiter
  - le nombre d'enregistrements
  - les interactions/la synchronisation liée aux sauvegardes
  - les volumes de données échangés/conservés

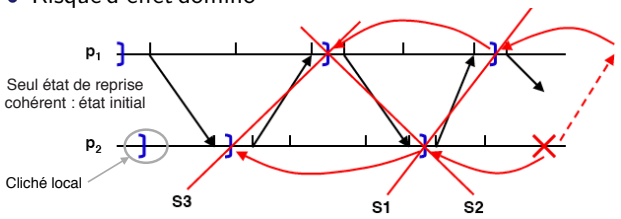
Difficultés

- L'enregistrement de sauvegardes est *local* à chaque site
- La cohérence est une propriété *globale*

Construction d'un état global cohérent

2 familles d'algorithmes

- *Collecte synchronisée* des états locaux à un même *instant logique*
  → algorithmes de prise de clichés (ex : Chandy-Lamport)
  - Les enregistrements locaux sont synchronisés pour que l'état gobal résultant soit cohérent
  - La synchronisation a un coût à l'exécution
    - une diffusion par site + algorithme de coordination de la collecte
- *Reconstruire* un état cohérent *à partir d'enregistrements locaux non coordonnés*
  - Traitement réalisé hors ligne (→ surcoût réduit à l'exécution)
  - Aucune interaction entre sites
  - Risque d'effet domino



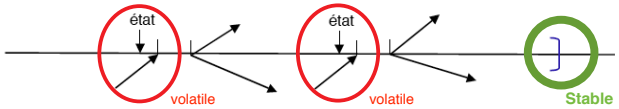
Construction de clichés à partir d'historiques locaux asynchrones

Principe

- chaque site enregistre ses clichés locaux périodiquement et indépendamment ;
- l'état global n'est construit qu'au moment de la reprise.

Enregistrement des clichés locaux : ingrédients de base

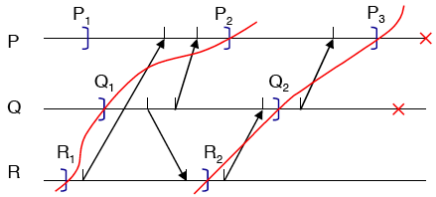
- pour chaque site, et chaque événement
  état + message reçu (éventuel)
- enregistrement
  - en continu, en mémoire volatile
  - périodique, en mémoire stable :
    copie des enregistrements en mémoire volatile effectués depuis le dernier enregistrement en mémoire stable (point de reprise)



Protocole d'enregistrement asynchrone (Juang-Venkatesan)

Sites orphelins

Un site est dit *orphelin* s'il a reçu un message dont l'émission n'a pas été enregistrée dans l'état courant.
→ son état ne peut faire partie d'une coupe cohérente.



Exemple

P_3, Q_2, R_2, ou P_2, Q_1, R_1
→ *effet domino*

Principe

Détecter les sites orphelins en comptant les messages émis et reçus

Validité d'un ensemble de points de reprise

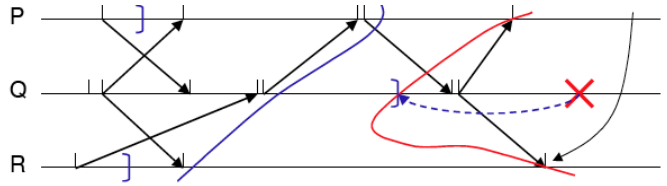
Hypothèses

- canaux fiables FIFO, asynchrones
- comportement d'un site : (réception ; traitement ; émission)*

Notations

- $R(PR_i)_{j\to i}$: nombre de messages reçus par i depuis j, enregistré au point de reprise PR_i du site i
- $E(PR_k)_{k\to l}$: nombre de messages envoyés par k vers l, enregistré au point de reprise PR_k du site k

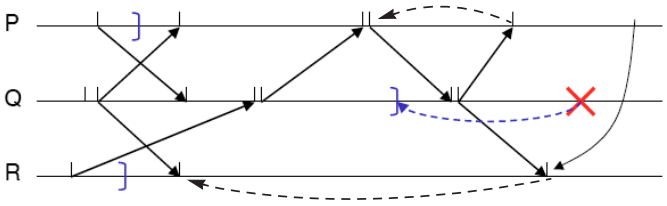
Un ensemble de points de reprise $(PR_s)_{s\in Sites}$ est *valide* ssi
$\forall i,j \in Sites : R(PR_i)_{j\to i} \leq E(PR_j)_{j\to i}$



Calcul d'un point de reprise global

En cas de reprise

- le site qui redémarre diffuse un message de reprise à tous
- chaque site vérifie que son dernier point de reprise vérifie la condition de validité.
  Sinon,
  - le site remonte au dernier point de reprise vérifiant la condition,
  - et diffuse un message de reprise à son tour (effet domino)

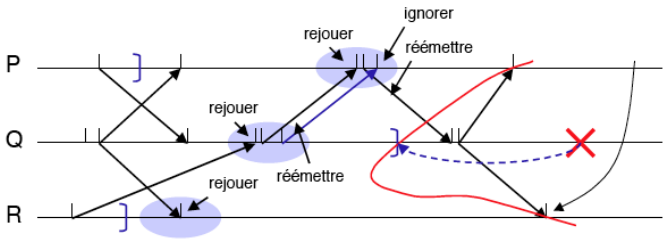


Réalisation de la reprise à partir d'un point de reprise valide

Difficulté : reconstituer (uniquement) les messages en transit

→

- *rejouer* le dernier message reçu (enregistré au point de reprise)
- rejouer le traitement
- *réémettre* les messages émis à la fin du traitement
- *ignorer* les messages *dupliqués* :
  - messages non en transit, émis et reçus juste avant la coupe (donc rejoués par le récepteur)
  - messages en transit au début de la reprise, et réémis à la reprise



Plan

Serveurs tolérants aux fautes : principes

Base : redondance → N serveurs pour résister à N-1 pannes franches

Cohérence du service fourni

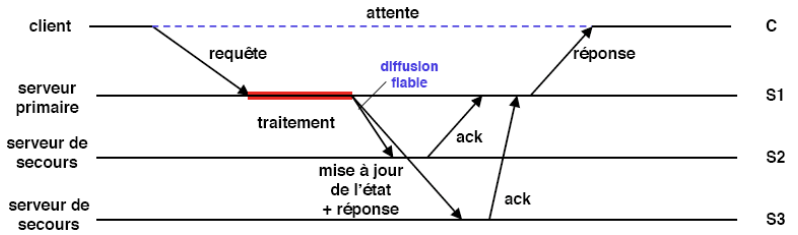
Pour le client, les N serveurs doivent se comporter comme un *même serveur*
→ Conditions *suffisantes* de cohérence

- Si *un* serveur traite une requête, *tous* les serveurs valides la traitent
- Tous les serveurs traitent toutes les requêtes dans le *même ordre*

Techniques de base

- *Redondance passive* (récupération)
  - Un serveur (*primaire*) traite (seul) les requêtes des clients
  - Les autres serveurs (*de secours*) scrutent (ping/pulsation) le primaire, pour prendre le relais en cas de défaillance.
- *Redondance active* (compensation)
  - N serveurs symétriques, qui exécutent *tous toutes* les requêtes

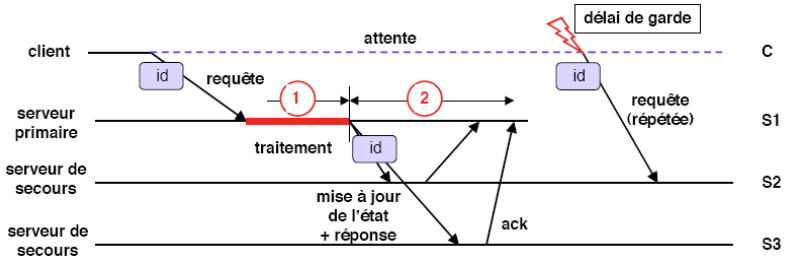
Redondance passive : serveur primaire-serveurs de secours
Protocole



Cohérence

- le primaire traite seul les requêtes et les traite en séquence → ordre global assuré sur les traitements
- diffusion fiable → quand le primaire renvoie la réponse, tous les serveurs corrects sont dans le même état
  → tout serveur de secours peut remplacer le primaire

Traitement de la défaillance du serveur primaire



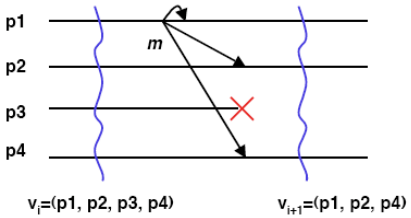
- *Détection* de la panne par le *client et* par les serveurs de *secours*
  - Les serveurs de secours élisent un nouveau primaire
  - Le client localise et contacte le nouveau serveur primaire
- diffusion fiable → tous les serveurs sont à jour, ou aucun
  - panne survenue en 1 (aucun serveur de secours n'est à jour) : le nouveau primaire traitera la requête comme une nouvelle requête
  - panne survenue en 2 (tous les serveurs de secours sont à jour) : le nouveau primaire renvoie simplement la réponse
  - Les requêtes ont un identifiant unique, pour distinguer entre 1 et 2

Utilisation des protocoles de groupes

vue $\triangleq$ (message contenant la) liste des serveurs valides

Après avoir traité une requête, le serveur primaire
- diffuse sa mise à jour aux serveurs secondaires (présumés corrects) de sa vue courante v_i
- construit la prochaine vue v_{i+1}, à partir des acquittements reçus, et des demandes d'intégration de nouveaux serveurs secondaires

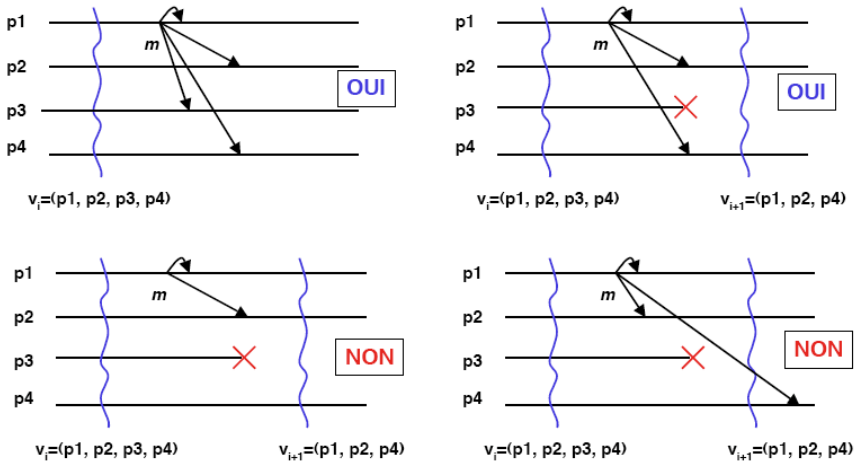
La diffusion fiable au sein de la vue assure que les processus présumés corrects et effectivement corrects sont tous à jour



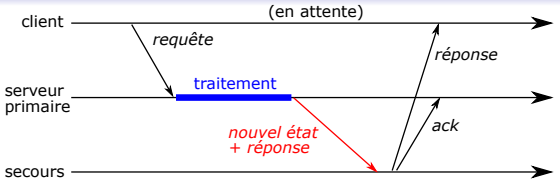
Soient
- m la mise à jour diffusée vers la vue v_i
- $P = v_i \cap v_{i+1}$

Alors m doit parvenir avant v_{i+1}
- ou bien à tous les membres de P
- ou bien à aucun membre de P

Vues synchrones : exemples

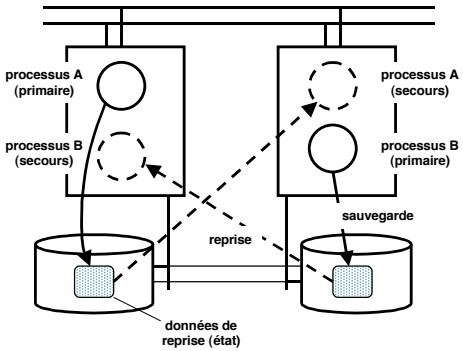


Cas particulier : un seul serveur de secours (Alsberg & Day)



- Un seul serveur de secours → protocole simplifié
  - mécanisme de vues et diffusion fiable inutiles
  - réponse envoyée directement le secours (cohérence garantie)
    → petit gain de temps par rapport au schéma général
- Panne du serveur primaire
  - détection par le client (délai de garde)
  - le serveur de secours devient primaire
  - primaire réinséré (comme secours) après réparation et copie de l'état
- Panne du serveur de secours
  - transparent pour le client
  - réinsertion après réparation (copie de l'état)
- Le schéma tolère la défaillance d'un serveur

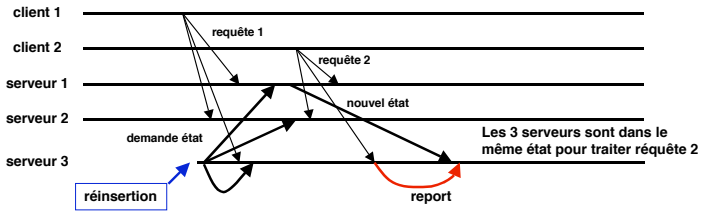
Exemple (schéma Alsberg & Day) : Tandem Non-Stop



- Sauvegarde périodique sur disque de l'état des processus.
- Surveillance mutuelle de chaque système (pulsations "je suis en vie")
- Si le délai de garde est dépassé, le processeur survivant reprend les processus défaillants à partir du dernier point de reprise enregistré.

Réinsertion après panne

- pour se réinsérer après une panne, un serveur diffuse *atomiquement* une demande de réinsertion
- un serveur recevant une demande de réinsertion transmet aussitôt son état courant

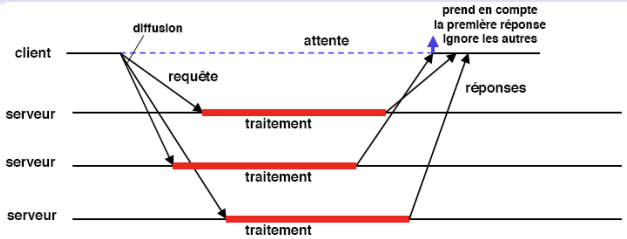


L'instant où la requête de réinsertion est délivrée permet de dater l'état transmis, par rapport à l'ensemble des requêtes traitées

Comportement du serveur en attente de réinsertion

- Une requête (1) délivrée *avant* la demande de réinsertion est *ignorée*
- Le traitement d'une requête (2) délivrée *après* la demande de réinsertion, est *retardé* jusqu'après l'arrivée du nouvel état

Redondance active
Protocole



Tous les serveurs sont équivalents et traitent les mêmes requêtes

Cohérence

Il suffit que la diffusion ait les propriétés suivantes

- tout message diffusé est reçu par tous les destinataires valides, ou par aucun
- tous les messages diffusés sont reçus partout dans le même ordre

→ propriétés définissant la *diffusion atomique*

Remarque : la contrainte d'ordre est nécessaire,
pour garantir que les serveurs évoluent tous de la même manière

Comparaison entre redondance passive et redondance active

Mécanismes nécessaires

- Redondance passive : protocole de groupe dynamique (vues synchrones, sauf si 1 seul serveur de secours, cas très courant)
- Redondance active : diffusion atomique
- Les deux mécanismes sont comparables (en complexité)

Usage

- Redondance passive
  - serveurs de secours disponibles pour d'autres tâches
  - le client doit détecter la panne du serveur primaire
  - reprise non immédiate
- Redondance active
  - tous les serveurs sont mobilisés
  - pas de retard en cas de panne
  - pannes transparentes pour les clients

Conclusion

- Redondance passive : le plus couramment utilisé
- Redondance active : applications critiques, temps réel
- Systèmes critiques : combinaison des différentes techniques, (p. ex. 2 serveurs actifs +1 serveur passif + points de reprise)