Tolérance aux fautes

Introduction : Objectifs, terminologie

Tolérance aux fautes

Mise en œuvre de la tolérace aux fautes
- Détection
- Récupération
- Compensation

Réalisation de serveurs à haute disponibilité
- Redondance passive
- Redondance active
- Exemples
- Groupes et diffusion

Tolérance aux fautes pour les données
- Techniques logicielles
- Techniques matérielles

Adapté de

Cours de systèmes répartis de Sacha Krakowiak (UJF), accessible en ligne

1 – Introduction

1) Définitions

Sûreté de fonctionnement (dependability)

Propriétés d'un système informatique définissant
les besoins des utilisateurs relativement à la *bonne exécution* du service délivré

Domaine de la *tolérance aux fautes*

- *Fiabilité* (*reliability*) : le service est rendu sans interruption
  Mesure : probabilité qu'il n'y ait pas de panne jusqu'à l'instant t
- *Disponibilité* (*availability*) : le service est opérationnel
  Mesure : fraction du temps durant laquelle le système fournit le service
- *Sécurité* ou *sûreté* (au sens *safety*) : un mauvais fonctionnement du système n'a pas d'incidence grave sur son environnement
  → définir « grave » et « environnement »
- *Sécurité* (au sens *security*) : contrôle de l'accès au service, confidentialité et intégrité des informations

Remarque

Le poids relatif des critères dépend de la nature de l'application, des besoins des utilisateurs…
- Système embarqué : fiabilité, disponibilité
- Système de communication (ex. : commutateur téléphonique) : disponibilité
- Service de fichiers, base de données : disponibilité, sécurité (security)
- Système de transport (ex. : navigation, guidage, freinage) : sûreté (safety), disponibilité

Défaillance (ou panne)

Un système (ou composant) est sujet à une *défaillance* (*failure*) lorsque son comportement n'est pas conforme à sa spécification

Degrés de permanence des défaillances

Les défaillances n'ont pas forcément un comportement uniforme dans le temps :
- Défaillance *transitoire* : se produit de manière isolée
- Défaillance *intermittente* : se reproduit sporadiquement
- Défaillance *permanente* : persiste indéfiniment (jusqu!à réparation) après son occurrence

Les défaillances non permanentes sont difficiles à observer, et donc à traiter

On peut définir différents degrés de gravité de défaillance (modèles de pannes)

Fonction des exigences des utilisateurs en termes de sûreté de fonctionnement :

- cas particuliers de pannes courantes pouvant être traitées efficacement (ex: pannes franches)
- pire cas : garantie de pouvoir traiter des pannes quelconques (difficile) (ex: pannes byzantines)

Modèles de pannes

Panne franche (ou : arrêt sur défaillance (*fail stop*))

Soit le système fonctionne, et donne un résultat correct, soit il est en panne (défaillant), et ne fait rien

Cas le plus simple, auquel on peut essayer de se ramener
(par exemple en forçant l'arrêt d'un composant dès qu'une erreur y est détectée : technique *fail fast*)

Pannes de temporisation

L'écart par rapport aux spécifications concerne uniquement le temps (tps de réaction à un événement...)

Panne par omission

Le système perd des messages entrants (omission en réception), sortants (omission en émission), ou les deux. Il n'y a pas d'autres déviations par rapport aux spécifications
- Ce modèle peut servir à représenter des défaillances du réseau, ou des pannes intermittentes
- Plus difficile à traiter que la panne franche

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 : hypothèse parfois nécessaire pour des systèmes à très haute fiabilité dans un environnement hostile (nucléaire, spatial…)
- Traitable, mais nécessite une redondance élevée
  (typiquement 3k+1 exemplaires d'un composant pour résister à k défaillances byzantines)

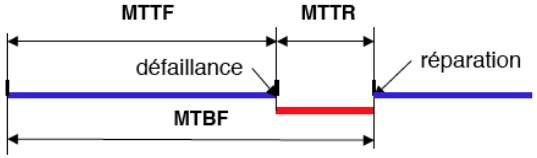
Mesures de fiabilité et disponibilité

Fiabilité

- MTTF (Mean Time To Failure) : Temps moyen jusqu'à la prochaine panne

Réparation

- Réparer un système en panne : le remettre en état de rendre un service correct
- Mesure : MTTR (Mean Time To Repair) : temps moyen de réparation
- MTBF (Mean Time Between Failures) = MTTF+MTTR

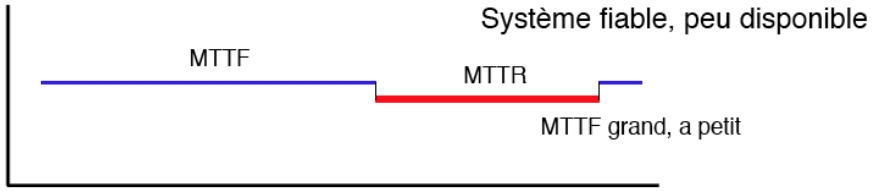


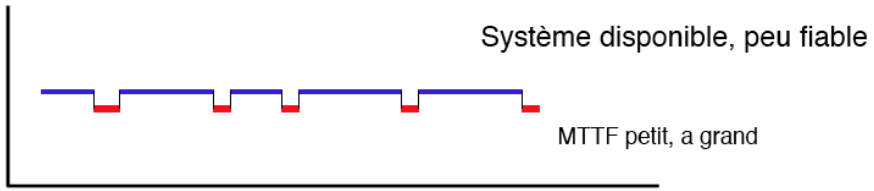
Disponibilité

- instantanée : probabilité que le système fonctionne à l'instant t
- moyenne : fraction moyenne du temps où le système est disponible (sur une période donnée)
- cas de pannes franches : dispo. moyenne = MTTF/(MTTF + MTTR), mesurée en *nombre de 9* :
  99.99% (4 nines) indisponible 50 minutes/an
  99.999% (5 nines) indisponible 5 minutes/an

Remarques

- Indisponibilité = 1 - disponibilité = MTTR/(MTTF + MTTR) ≈ MTTR/MTTF (si MTTR << MTTF)
  → diviser MTTR par 2 a le même effet sur la disponibilité que doubler MTTF
- Différence entre fiabilité (mesure : MTTF) et disponibilité (mesure : a = MTTF/(MTTF+MTTR))





2) Analyse des pannes

Définitions

Erreur

(partie de l') état du système *susceptible de* provoquer une panne
→ partie de l'état interne dont les propriétés courantes ne sont pas conformes aux spécifications

- exemples
  ◊ logiciel : l'indice courant d'un tableau est supérieur à sa dimension
  ◊ matériel : une connexion est coupée entre deux points qui devraient être reliés

Faute

Toute cause (événement, action, circonstance) *pouvant* provoquer une erreur

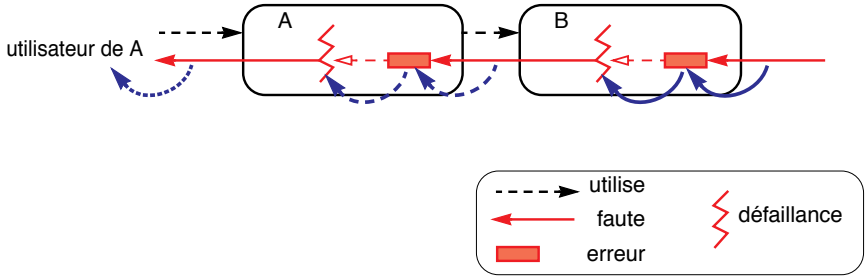
- exemple : faute de programmation, malveillance, catastrophe naturelle, etc.

De l'erreur à la défaillance

- Une erreur est *susceptible d'*entraîner une panne, mais ne la provoque pas forcément
  ◊ parce qu'il y a assez de redondance interne pour que le système reste opérationnel
  ◊ parce que la partie erronée de l'état n'est pas utilisée par les fonctions en cours d'exécution
- Une erreur est *latente* tant qu'elle n'a pas provoqué de défaillance
- Le temps entre l'apparition de l'état d'erreur et la défaillance est le *délai de latence*
  Plus le délai de latence est long, plus la recherche des causes d'une défaillance est difficile

Propagation entre composants (ou sous-systèmes)

Un composant A utilise un composant B si le bon fonctionnement de A dépend de celui de B



Pour A, la défaillance de B (service incorrect) constitue une faute,
qui peut à son tour provoquer une erreur interne à A, puis une défaillance de A

faute → erreur → défaillance → faute → erreur → …

2 – Tolérance aux fautes

1) Comment garantir la sûreté de fonctionnement ?

Évitement des fautes

essayer d'empêcher les fautes de se produire

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

Tolérance aux fautes :

préserver le service malgré l'occurrence de fautes

- Moyen : *redondance*
  - ◊ redondance d'information (détection d'erreur)
  - ◊ redondance temporelle (traitements multiples)
  - ◊ redondance matérielle (composants dupliqués)

2) Réalisation de la tolérance aux fautes

Constat de départ

la tolérance aux fautes est indispensable pour la sûreté de fonctionnement

Quelles que soient les précautions prises, l'occurrence de fautes est inévitable (erreur humaine, malveillance, vieillissement du matériel, accident naturel…).

→ les mesures prises pour prévenir ou d'éliminer les fautes peuvent seulement réduire la probabilité de leur occurrence (ce qui est essentiel), mais non les éliminer totalement

→ Il faut donc concevoir les systèmes de manière à ce qu'ils continuent à rendre le service attendu (éventuellement de manière dégradée), même en présence de fautes

Remarque

Il n'existe pas de méthode efficace et universelle : les méthodes disponibles sont lourdes, coûteuses, et spécifiques à un modèle de fautes donné. Le contexte de mise en œuvre doit donc être bien analysé et explicité.

Difficulté de la tolérance aux fautes

Les pièges sont nombreux …

- Mauvaise analyse des fautes possibles
  - ◊ ne pas prévoir des cas possibles de faute
  - ◊ laisser passer des fautes sans réaction
- Réaction inappropriée due à des hypothèses mal formulées
  - ◊ le « remède » peut aggraver la situation
- Mise en oeuvre incorrecte de la redondance
  - ◊ les éléments redondants doivent avoir des modes de défaillance différents
  - ◊ une redondance inadaptée (mal utilisée) donne un faux sentiment de sécurité

Deux exemples d'école

- Ariane 501
  http://www.cnes.fr/espace_pro/communiques/cp96/rapport_501/rapport_501_2.html
- Therac 25 (radiothérapie)
  http://sunnyday.mit.edu/therac-25.html

3 – Mise en œuvre de la tolérance aux fautes

Deux techniques de base

Récupération (error recovery)

- Détecter l'erreur (identifier la partie incorrecte de l'état), au moyen d'un test de vraisemblance (→ redondance)
  - ◊ explicite (exprimer et vérifier des propriétés spécifiques de l'état)
  - ◊ implicite (observation d'anomalies : violation d'un délai de garde, de protection mémoire…)
- Remplacer l'état d'erreur par un état correct
- Deux techniques de récupération
  - ◊ Reprise (à partir d'un état ancien enregistré)
  - ◊ Poursuite (reconstitution d'un état courant correct)
- Exemples : Tandem Non-Stop, points de reprise dans un système réparti

Compensation (error masking)

- Le système a une redondance interne suffisante pour détecter et corriger l'erreur (→ tests de vraisemblance externes inutiles) → traitement d'erreur transparent pour les utilisateurs
  - ◊ composants dupliqués, pour réduire la probabilité qu'une faute amène une défaillance
  - ◊ réaliser des traitements multiples
    Exemple : réaliser la même opération avec des algorithmes/des matériels différents
- Exemple : vote majoritaire (Tandem Integrity)

1) Détection d'erreur

Objectifs

- prévenir (si possible) l'occurrence d'une défaillance provoquée par l'erreur
- éviter la propagation de l'erreur à d'autres composants
- faciliter l'identification ultérieure de la faute (en vue de son élimination ou de sa prévention)

Paramètres

- *latence* : délai entre production et détection de l'erreur
- *taux de couverture* : pourcentage d'erreurs détectées

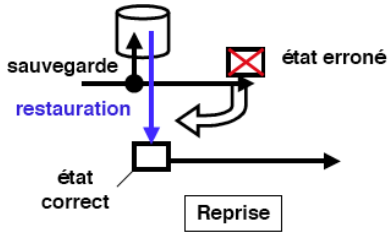
Techniques

- Comparaison des résultats de composants dupliqués
  - ◊ coût élevé
  - ◊ latence faible
  - ◊ taux de couverture élevé
  - ◊ nécessité d'indépendance des conditions de création et d'activation des fautes (cf Ariane 5)
- Contrôle de vraisemblance
  - ◊ coût modéré
  - ◊ latence variable selon la technique
  - ◊ taux de couverture souvent faible

2) Récupération

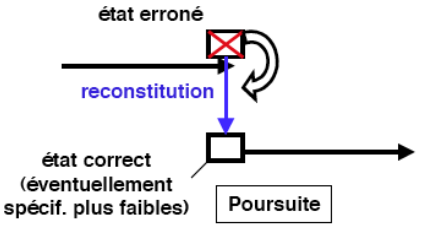
Reprise (backward recovery)

- Retour en arrière vers un état antérieur dont on sait qu'il est correct
- Nécessite la sauvegarde de l'état
- Technique générale



Poursuite (forward recovery)

- Tentative de reconstitution d'un état correct, sans retour arrière
- La reconstitution est souvent seulement partielle, d'où service dégradé
- Technique spécifique, la reconstitution dépend de l'application, et nécessite une analyse préalable des types d'erreurs possibles



Exemple

Communication par paquets : comment réagir à la perte d'un paquet ?

Reprise

- L'émetteur conserve une copie des paquets qu'il a envoyés, jusqu'à recevoir un acquittement
- En cas de perte détectée (délai de garde), le récepteur demande la réémission du paquet manquant

Poursuite

- Pour un type particulier d'application, l'émetteur peut reconstituer (totalement ou en partie) le contenu d'un paquet manquant en utilisant les paquets voisins (suivants, précédents)
- Le récepteur peut alors poursuivre sans interroger l'émetteur

Comparaison

- Reprise : technique générique, la plus utilisée ; peut être coûteuse en temps et place
- Poursuite : technique spécifique, peut être efficace, mais d'application limitée
- Dans la suite, nous ne traitons que de la reprise

Techniques de 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 de la reprise

- sauvegarde et restauration doivent être atomiques (pas d'interférences)
- l'état des points de reprise doit lui-même être protégé contre les fautes
- dans un système réparti :
  - ◊ les points de reprise sont créés pour chaque processus
  - ◊ l'ensemble de ces points de reprise doit constituer un état global cohérent du système
  - → garantir/évaluer la coordination de la prise des clichés sur chacun des sites (vu ultérieurement)

Exemple de récupération par reprise : Tandem Non-Stop

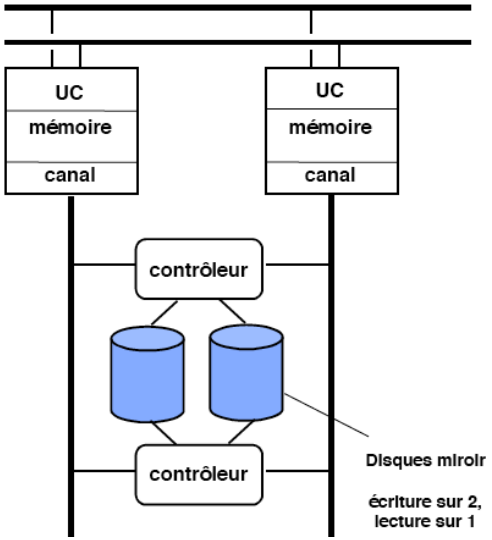
Tolère une faute unique d'un composant (UC, bus, mémoire, contrôleur, disque)

Matériel

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

Logiciel

- Paires de processus (processus actif - processus de secours)
- Système d'exploitation spécifique (Unix modifié pour gestion des paires de processus)

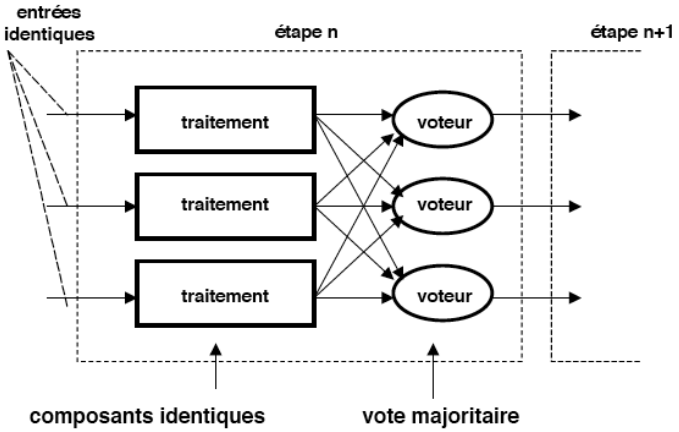


3) Compensation

But : permettre l'exécution d'un traitement *sans altérations*, en présence de fautes

Exemple : vote majoritaire

Tandem Integrity : système TMR (Triple Modular Redundancy)



Résiste à la défaillance d'un composant de traitement et d'un voteur par étape

Compensation : bilan

Avantages

- Traitement d'erreur rapide
- Masquage : défaillances internes invisibles aux composants utilisateurs, à condition que
  - ◊ les hypothèses de fautes soient respectées
  - ◊ les fautes soient indépendantes sur les composants dupliqués

Inconvénients

- Redondance élevée, donc coût élevé
  - ◊ Tandem Integrity (TMR) : X 3

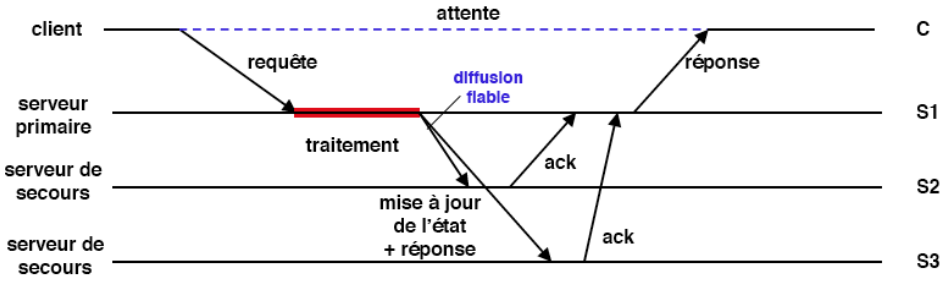
4 – Réalisation de serveurs à haute disponibilité

1) Serveurs tolérants aux fautes : principes

- La tolérance aux fautes est obtenue par *redondance*
  - ◊ Hypothèse : pannes franches (fail stop)
  - ◊ On utilise N serveurs pour résister à la panne de N—1 serveurs
- La redondance impose des contraintes de cohérence
  - ◊ Vu du client, l'ensemble des N serveurs doit se comporter comme un serveur unique
    → Conditions *suffisantes* de cohérence
    - Si une requête est traitée par un serveur, elle doit l'être par tous les serveurs valides
    - Deux requêtes différentes doivent être traitées dans le même ordre par tous les serveurs
- Techniques de base
  - ◊ Récupération : redondance passive (schéma serveur primaire - serveurs de secours)
    - Un serveur (primaire) joue un rôle privilégié
    - La panne du primaire est visible par les clients (changement de serveur)
    - La panne d'un serveur de secours est invisible aux clients
  - ◊ Compensation : redondance active (compensation)
    - Les N serveurs jouent un rôle symétrique et exécutent tous les mêmes requêtes
    - Les pannes sont invisibles aux clients tant qu'il reste un serveur en marche

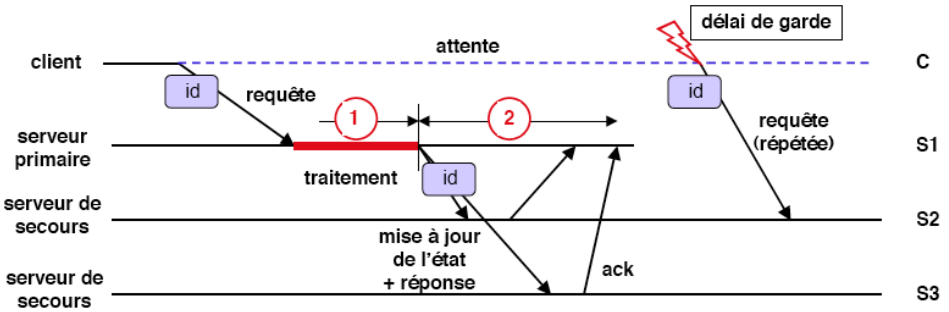
2)Redondance passive: serveur primaire-serveurs de secours

Protocole



- Hypothèse : diffusion fiable (pas de diffusion partielle) :
  le message parvient à tous ses destinataires non en panne, ou à aucun
  → Cohérence :
  Quand le primaire renvoie la réponse, tous les serveurs non en panne sont dans le même état
- Récupération : tout serveur de secours peut remplacer le primaire en cas de panne

Traitement des défaillances

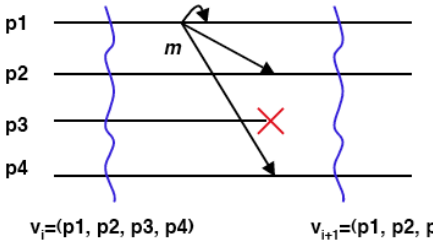


- Une défaillance du primaire *doit* être détectée par le client *et* par les serveurs de secours
  - ◊ Les serveurs de secours choisissent (« *élisent* ») un nouveau serveur primaire
  - ◊ Le client localise et contacte le nouveau serveur primaire
- Grâce à la diffusion fiable, tous les serveurs de secours sont à jour, ou aucun.
  - ◊ Si la panne est arrivée en 1 (aucun serveur de secours n'est à jour) :
    le nouveau primaire réexécute la requête et met à jour les serveurs de secours
  - ◊ Si la panne est arrivée en 2 (tous les serveurs de secours sont à jour) :
    le nouveau primaire renvoie simplement la réponse
  - ◊ Les requêtes ont une identification unique, ce qui permet de distinguer entre 1 et 2

Mise en œuvre de la cohérence : communication de groupe

Outil de base : la *vue*, message contenant la liste des processus valides

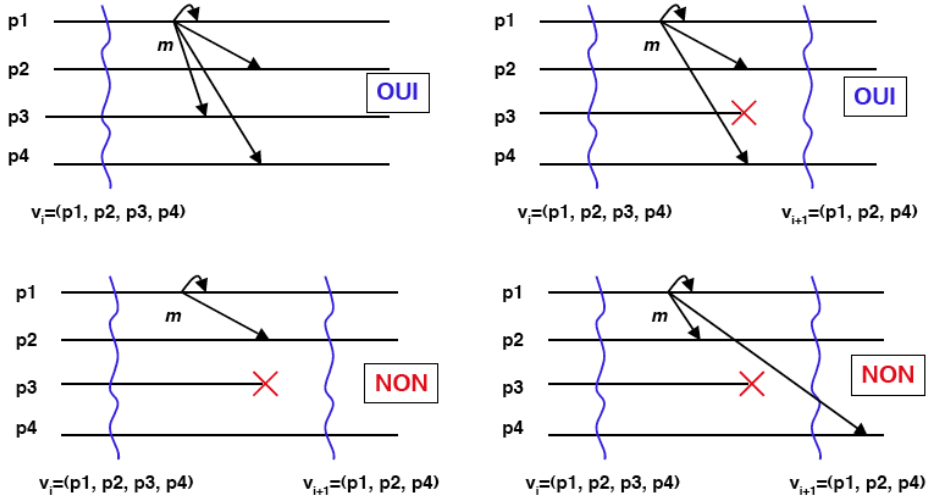
- Le protocole de redondance passive repose en effet sur la connaissance (la *vue*) par le serveur primaire de l'ensemble (ou *groupe*) des serveurs secondaires valides (non en panne):
  Après avoir traité une raquête, le serveur primaire
  - diffuse sa mise à jour aux serveurs secondaires (présumés 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
- Propriété de cohérence : cohérence entre évaluation des vues et diffusion des messages
  Les processus présumés corrects et effectivement corrects doivent *tous* être à jour



Soient
- m la mise à jour diffusée dans 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

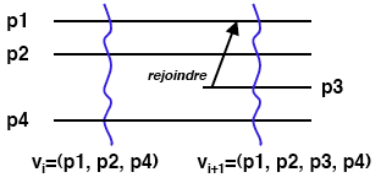
Vues synchrones : exemples



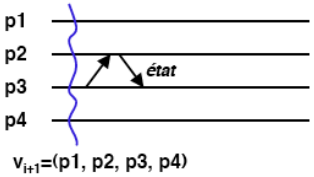
Utilisation du mécanisme de groupe pour la réinsertion après panne

Après réparation, le serveur défaillant doit se réinsérer et restaurer son état

1 - le serveur qui se réinsère envoie un message *rejoindre* → changement de vue

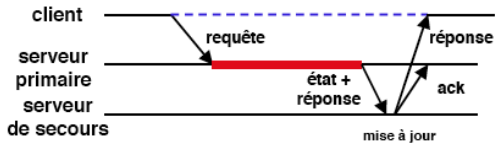


2 - le serveur réinséré demande une copie de l'état à l'un des autres serveurs



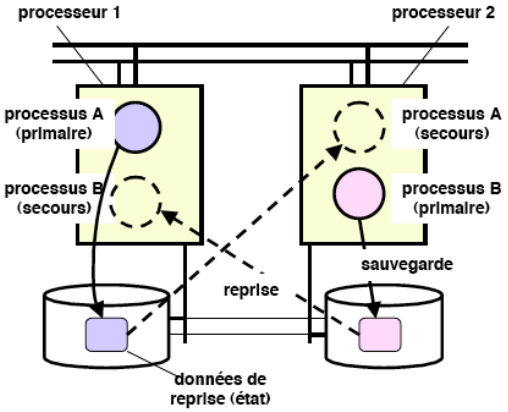
3 - Le serveur réinséré peut fonctionner (répondre aux requêtes)

Cas particulier de deux serveurs (schéma de Alsberg & Day)



- L'ensemble des serveurs de secours est un singleton → protocole simplifié
  - ◊ Réponse envoyée directement par le serveur de secours (cohérence garantie) → petit gain de temps par rapport au schéma général
  - ◊ Mécanisme de vues inutile
- Le serveur de secours surveille le primaire
- En cas de panne du serveur primaire
  - ◊ Détection par le client (délai de garde)
  - ◊ Le serveur de secours devient primaire
  - ◊ Réinsertion du primaire (comme serveur de secours) après réparation, avec copie de l'état
- En cas de panne du serveur de secours
  - ◊ Rien à faire par le client
  - ◊ Réinsertion après réparation (copie de l'état)
- Le schéma tolère la défaillance d'un serveur

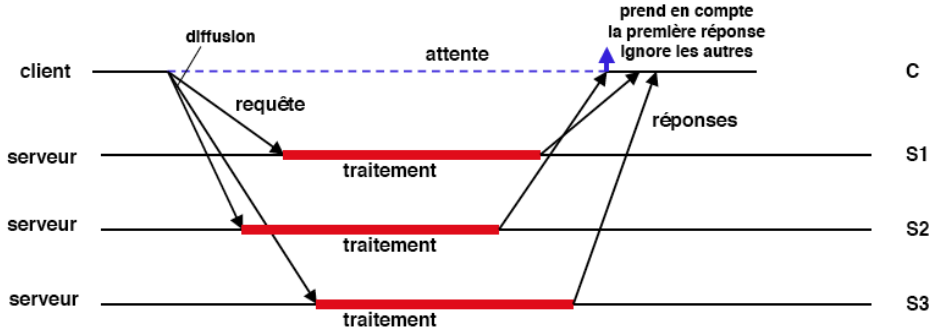
Exemple : Tandem



- Sauvegarde périodique sur disque de l'état des processus.
- Surveillance mutuelle de chaque système par l'autre (messages "je suis vivant").
- En cas de dépassement du délai de garde, le processeur survivant reprend les processus défaillants à partir du dernier point de reprise enregistré.

3)Redondance active

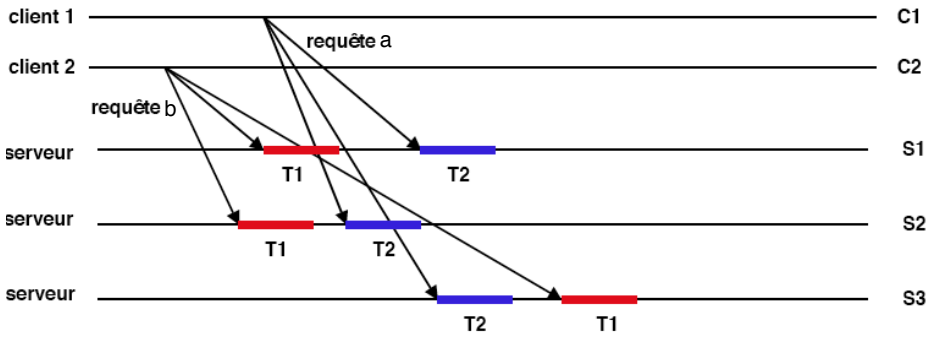
Protocole



- Tous les serveurs sont équivalents et exécutent le même traitement
- La panne d'un serveur est invisible aux clients
- Pour garantir la cohérence, il suffit que la diffusion ait les propriétés suivantes
  - ◊ Un message diffusé est reçu par tous les destinataires (non en panne), ou par aucun
  - ◊ Deux messages différents sont reçus dans le *même ordre* par tous les destinataires

  Ces propriétés définissent la *diffusion atomique* (ou : *diffusion totalement ordonnée*)

Nécessité de la diffusion atomique



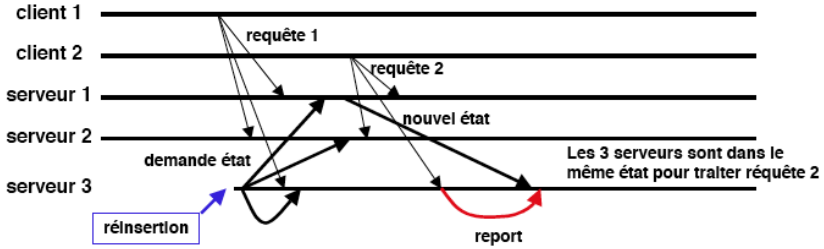
Dans l'exemple ci-dessus, la diffusion n'est pas atomique

- Serveur 1 : T1 ; T2
- Serveur 2 : T1 ; T2
- Serveur 3 : T2 ; T1

Si les traitements modifient l'état des serveurs, le système est incohérent

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 immédiatement son état courant



- *Propriété essentielle* : 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 nouveau serveur
  - ◊ Un message tel que `requête 1` (parvenant avant sa propre demande d'état) est ignoré
  - ◊ Le traitement des messages comme `requête 2` (parvenant avant la réponse à la demande d'état) est retardé jusqu'après l'arrivée du nouvel état, afin de garantir la cohérence, grâce aux propriétés de la diffusion atomique

Comparaison entre serveur primaire et redondance active

- Mécanismes nécessaires
  - ◊ Serveur primaire : technique de groupe dynamique (vue synchrones, sauf si 1 seul serveur de secours, cas très courant)
  - ◊ Redondance active : diffusion atomique
  - ◊ Les deux mécanismes sont équivalents (en complexité)
- Usage des ressources
  - ◊ Serveur primaire : les serveurs de secours peuvent exécuter des tâches de fond ; reprise non immédiate
  - ◊ Redondance active : tous les serveurs sont mobilisés, reprise immédiate
- Travail pour les clients
  - ◊ Serveur primaire : le client doit détecter la panne du primaire
  - ◊ Redondance active : le client n'a rien à faire
- Conclusion
  - ◊ Serveur primaire : le plus utilisé dans les applications courantes
  - ◊ Redondance active : applications critiques, temps réel

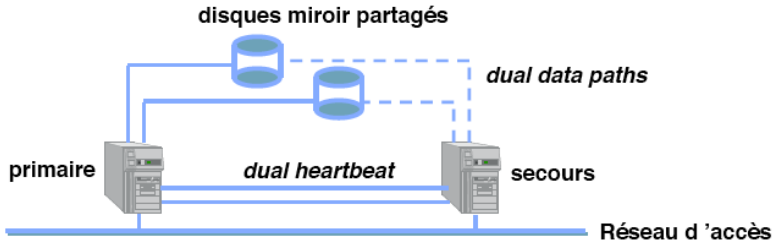
4)Serveurs à haute disponibilité

Ingrédients

- Duplication passive des serveurs : primaire et secours
- Réseaux : réseau d'accès, réseaux heartbeat (émission d'un message à intervalles réguliers)
- Disques :
  - ◊ Disques privés (par serveur)
  - ◊ Disques partagés (redondants → RAID miroir ou parité)
- Pas de point de défaillance unique (single point of failure) : redondance des composants dont la panne entraînerait la panne du système dans son ensemble
  - ◊ Alimentation électrique
  - ◊ Événements et services externes, équipements réseau…

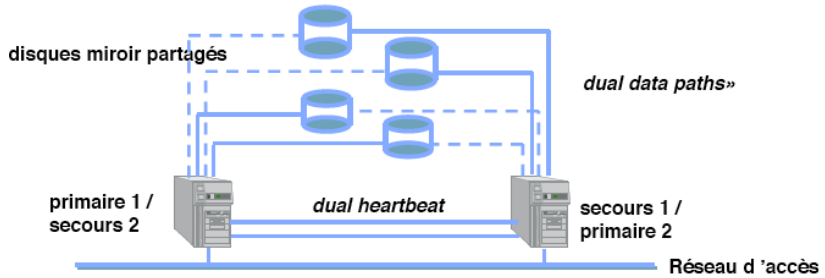
Configurations

Configuration 1-1 asymétrique



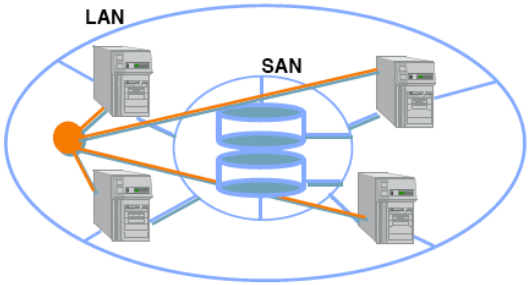
- Réseaux heartbeat dédiés
- Les disques partagés contiennent l'état du système
- Bascule (failover) entre primaire et secours en cas de panne du primaire
  L'accès aux disques partagés bascule également

Configuration symétrique



- Les serveurs restent indépendants l'un de l'autre
  → Moindre coût
- Impact sur la performance en cas de bascule
- Peut être généralisée à plusieurs applications (service level failover)

Serveurs en grappes (server farms)

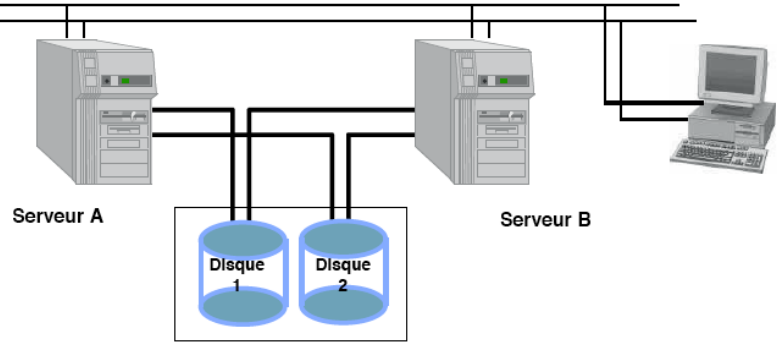


- Accès aux disques miroir via un SAN
  (Storage Area Network : espace disque (≠système de fichiers) partagé/accessible via le réseau)
- Réseaux heartbeat séparés
- Flexibilité accrue

Exemple de système à haute disponibilité : IBM HA-CMP

HA-CMP : High Availability Cluster MultiProcessing

Principe : la communication (pour reprise) se fait toujours via un disque partagé.



- *Récupération simple* : l'application fonctionne sur l'un des systèmes (primaire).
  L'autre système est inactif ou exécute des tâches de fond, non critiques.
- *Récupération mutuelle* : les applications sont exécutées sur les deux systèmes, sans partage de données. En cas de panne d'un système, ses applications sont reprises par l'autre.
- *Partage de charge* : les applications sont exécutées sur les deux systèmes et partagent des données, sous le contrôle d'un gérant distribué de verrous (service transactionnel)

5) Groupes et diffusion

Groupes

Opérations sur les groupes de processus

- appartenance (group membership)
  - ◊ évolution de la composition du groupe : ajout/retrait de membres
  - ◊ connaissance de la composition courante
- communication vers l'ensemble des membres du groupe : *diffusion* (*broadcast*/*multicast*)

La composition d'un groupe peut changer soit par entrée ou sortie volontaire soit par suite de défaillances ou réinsertions ; certaines spécifications restreignent les évolutions possibles

Utilisation

- gestion d'un ensemble de processus devant se comporter comme un processus unique : tolérance aux fautes (groupe de serveurs, gestion de données dupliquées, systèmes sur grappes de machines…)
- travail coopératif, partage d'information : chat, simulations en réseau

Diffusion

Définitions

La *diffusion* est un mode de communication dans lequel un processus émetteur envoie un message à un *ensemble* de processus destinataires.

- *Diffusion générale* (*broadcast*) :
  - ◊ les destinataires sont les processus d'*un seul ensemble* défini implicitement (connu ou non)
  - ◊ L'émetteur est également destinataire.
  - ◊ Exemples :
    - les membres d'un groupe unique, vus de l'intérieur du groupe ;
    - "tous" les processus du système
- *Diffusion de groupe* (*diffusion sélective, multicast*)
  - ◊ les destinataires sont les membres d'un ou plusieurs groupe(s) spécifié(s) et identifié(s); les groupes pouvent ne pas être disjoints.
  - ◊ L'émetteur peut ne pas appartenir au(x) groupe(s) destinataires(s)

Les primitives sont notées broadcast (p, m) et multicast (p, m, {g})

p : émetteur

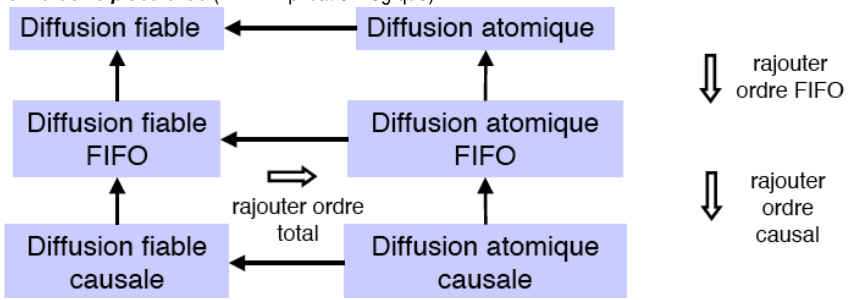
m : message

{g} : ensemble des groupes destinataires

Propriétés de la diffusion

- Propriétés indépendantes de l'ordre d'émission (concernent uniquement les récepteurs).
  - ◊ *Diffusion fiable* : un message est délivré à tous ses destinataires ou à aucun
  - ◊ *Diffusion totalement ordonnée* (ou *atomique*) : la diffusion est *fiable et* les messages sont délivrés dans le *même ordre* à tous leurs destinataires
- Propriétés liées à l'ordre d'émission
  - ◊ Diffusion *FIFO* : deux messages issus du *même émetteur* sont délivrés à tout récepteur dans leur ordre d'émission
  - ◊ Diffusion *causale* : l'ordre de réception des messages respecte leur ordre causal d'émission

Combinaisons possibles (→ = implication logique)

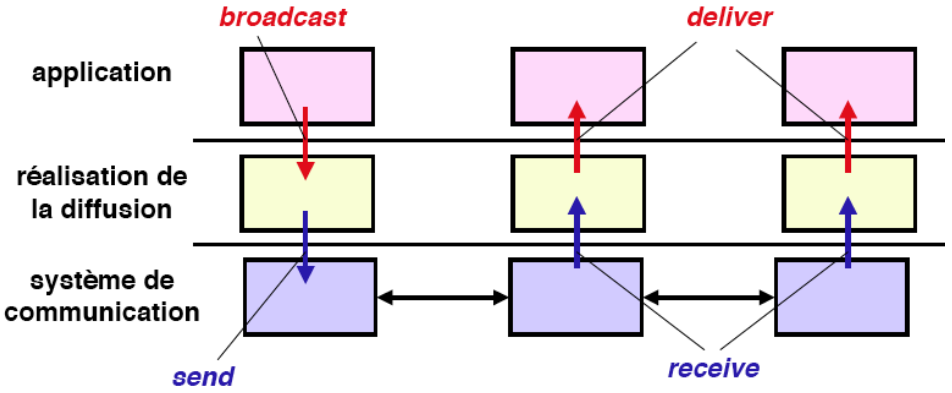


On peut en outre ajouter des propriétés de temporisation (diffusion dans un délai fixé)

Réalisation de la diffusion : situation

La diffusion est réalisée au-dessus d'un service de communication permettant l'envoi et la réception de messages (primitives *send* et *receive*).

Les primitives de la diffusion sont *broadcast* et *deliver*



Réalisation de la diffusion : hypothèses

On suppose que le service de communication est *fiable* (tout message finit par arriver, intact, s'il existe un lien physique entre émetteur et récepteur).

Par ailleurs, le service de communication sous jacent peut être :

- synchrone : le délai de transmission des messages a une borne connue
- asynchrone : pas de borne supérieure pour le délai de transmission (exemple : l'Internet)

  Dans ce cas, on ne peut jamais déterminer si un processeur qui ne répond pas est en panne, ou si la communication avec lui est retardée.

Résultats

- La diffusion fiable est réalisable simplement, même dans le cas asynchrone
- La diffusion atomique (totalement ordonnée) n'est pas réalisable en asynchrone (démontré).

Tous les algorithmes pratiques de réalisation de la diffusion atomique
font donc une hypothèse de synchronisme (et utilisent un délai de garde)

Réalisation de la diffusion causale

Utilisation de vecteurs d'horloges (Fidge-Mattern), vu précédemment

Réalisation de la diffusion fiable

Réalisation de *broadcast(p,m)* :

```
estampiller m avec sender(m) (= p, processus émetteur de m)
pour tous les voisins de p, et p faire
  send(m)
fin pour
```

Contrôle de *deliver(m)* :

Réalisation de de *receive(m, sender(m))* par le processus q :

```
si q n'a pas précédemment exécuté deliver(m) alors
  si sender(m) != q alors
     pour tous les voisins de q faire
        send(m)
     fin pour
  fin si
  deliver(m)
fin si
```

Ce protocole tolère les pannes franches tant que la communication reste possible entre processus corrects

La fiabilité est alors garantie car tout processus qui délivre un message l'a au préalable envoyé à ses voisins. Pour qu'un processus correct ne recoive pas un message, il faudrait donc qu'aucun processus correct ne l'ait envoyé, donc délivré.

Réalisation de la diffusion totalement ordonnée

Plusieurs méthodes sont possibles (l'ordre peut être imposé soit à l'émission soit à la réception)

- Utiliser un site *séquenceur*.
  - ◊ Pour diffuser un message, on l'envoie au séquenceur.
  - ◊ Celui-ci lui attribue un numéro (estampille), dans une suite croissante (1, 2, 3, etc.), et l'envoie à tous les destinataires.
  - ◊ Les destinataires délivrent les messages diffusés dans l'ordre croissant des estampilles.
  - ◊ Utilisé dans le protocole JavaGroups
  - ◊ Problème : résister à la panne du séquenceur
- Ordre utilisant des horloges logiques (horloges de Lamport) en réception.
  - ◊ Chaque message m est estampillé provisoirement par son heure logique de réception.
  - ◊ Les différents récepteurs se communiquent leurs estampilles provisoires (l'émetteur peut jouer le rôle de collecteur des estampilles provisoires)
  - ◊ Quand toutes sont connues, on attribue définitivement à m la plus grande.
    → tout message a la même estampille (définitive) sur tous les sites récepteurs.
    → l'estampille définitive est un majorant de l'estampille provisoire
  - ◊ Un message estampillé définitivement par e est délivré lorsque e est minimal parmi toutes les estampilles (provisoires ou définitives) du site.
    → Les messages sont délivrés dans l'ordre croissant des estampilles.
  - ◊ Principe utilisé dans la version initiale d'ABCAST (Isis).
  - ◊ Problème : le traitement des défaillances est complexe

Etude de cas : JavaGroups

Références

- site du projet JGroups (en anglais) `http://www.jgroups.org/index.html`
- Fournit la documentation (API, manuels, tutoriels, exemples), et les liens de téléchargement
- des transparents de présentation du service (en anglais), par l'auteur principal du projet `http://education.sys-con.com/node/84651`
- page wiki JGroups : `http://community.jboss.org/wiki/JGroups`

Objectif

fournir un service de diffusion plus élaboré que la diffusion IP :
diffusion fiable, ordonnée, causale, sélective…

Principe

- La communication se fait via un canal (un canal est associé à un groupe et un seul)
- Les propriétés requises pour la communication sont réalisées par des protocoles
  - ◊ chaque protocole assure une propriété particulière : fiabilité, ordonnancement, groupes
  - ◊ l'ensemble des propriétés souhaitées pour la communication définit l'ensemble de protocoles associé au canal
    - chaque protocole est implémenté par une classe java
    - les protocoles utilisés pour un canal sont structurés en une pile de protocoles
    - un fichier XML permet de spécifier et paramétrer chacun des protocoles constituant la pile associée à un canal

Opérations sur un canal

- **création/destruction** : `ch = new JChannel(fichier_protocole.xml)/ch.close()`
- **rejoindre un groupe** utilisant le canal : `ch.connect("nomGroupe")`
  le premier processus rejoignant le groupe crée le groupe, s'il n'existait pas
- **quitter** un groupe : `ch.disconnect()`
- **diffuser** un message : `ch.send(message)`
- **recevoir** (délivrer) un message.
  → synchrone : méthode `ch.receive(message)` associée au canal
  → ou via une interface de rappel :
  - le récepteur doit hériter de la classe `Receiver` ou `ReceiverAdapter`
  - ce qui implique qu'il implante une méthode `receive(message)`, appelée à la délivrance
  - le récepteur s'abonne auprès du canal par `ch.setReceiver(this)`

Autres méthodes de rappel

- un récepteur doit aussi implanter la méthode `viewAccepted(groupe_courant)`, appelée par le service de diffusion chaque fois que le groupe évolue
- deux méthodes permettent de communiquer aux membres l'état d'un objet associé au groupe: `membre.getState()` et `membre.setState(état)`. [Utile lorsque qu'un membre rejoint un groupe] pour être mis à jour, un processus doit appeler `canal.getState()`, ce qui provoquera le transfert de l'état (obtenu d'un autre membre par `mb.getState()`), par le rappel de `mb.setState()`
  ◊ l'appel de `m.setState()`permet d'ordonner la réception de l'état par rapport aux diffusions
  ◊ l'état du groupe est dupliqué sur les différents membres

Exemple : réalisation d'un chat (source : tutoriel Jgroups)

```
import org.jgroups.JChannel;
import org.jgroups.Message;
import org.jgroups.ReceiverAdapter;
import org.jgroups.View;
import org.jgroups.util.Util;

import java.util.List;
import java.util.LinkedList;
import java.io.BufferedReader;
import java.io.InputStreamReader;

public class SimpleChat extends ReceiverAdapter {
    JChannel channel;
    String user_name=System.getProperty("user.name", "n/a");
    final List<String> state=new LinkedList<String>();

    public void viewAccepted(View new_view) {
        System.out.println("** view: " + new_view); }

    public void receive(Message msg) {
        String line=msg.getSrc() + ": " + msg.getObject();
        System.out.println(line);
        synchronized(state) { state.add(line); }
    }

    public byte[] getState() {
        synchronized(state) {
            try { return Util.objectToByteBuffer(state);}
            catch(Exception e) {e.printStackTrace(); return null;
            }
        }
    }
```



```
    public void setState(byte[] new_state) {
        try {
            List<String> list=(List<String>)Util.objectFromByteBuffer(new_state);
            synchronized(state) { state.clear(); state.addAll(list);}
            System.out.println("rcvd state ("+list.size()+" msgs in chat history):");
            for(String str: list) { System.out.println(str);}
        } catch(Exception e) {e.printStackTrace();}
    }

    private void start() throws Exception {
        channel=new JChannel();
        channel.setReceiver(this);
        channel.connect("ChatCluster");
        channel.getState(null, 10000);
        eventLoop();
        channel.close();
    }

    private void eventLoop() {
        BufferedReader in=new BufferedReader(new InputStreamReader(System.in));
        while(true) {
            try {
                System.out.print("> "); System.out.flush();
                String line=in.readLine().toLowerCase();
                if(line.startsWith("quit") || line.startsWith("exit")) { break;}
                line="[" + user_name + "] " + line;
                Message msg=new Message(null, null, line);
                channel.send(msg);
            } catch(Exception e) {}
        }
    }

    public static void main(String[] args) throws Exception {
        new SimpleChat().start();
    }
}
```


Bilan

- Interface simple
- Logiciel libre
- Nombreux protocoles annoncés (mais pas tous effectifs)
  ◊ transport (UDP, TCP)
  ◊ découverte (Ping...)
  ◊ une forme de fiabilité, FIFO
  ◊ sécurité
  ◊ détection de pannes
  ◊ fragmentation
  ◊ transfert d'état
  ◊ trace
  ◊ diffusion probaliste
  ◊ synchronisme virtuel...
- Facilité de composition
- Protocoles/services élémentaires
- Communauté toujours active, mais assez réduite
- Développement irrégulier, en cours
- Pas toujours fiable
- Pas/peu (encore ?) de protocoles de haut niveau (diffusion ordonnée, causale), mais les outils/ protocoles élémentaires pour les réaliser

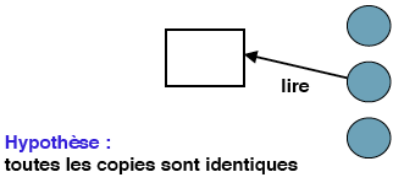
5 – Tolérance aux fautes pour les données

- Objectif : assurer la disponibilité des données
  - ◊ en présence de défaillances (matérielles ou logicielles)
  - ◊ le principe est toujours la redondance
- Techniques logicielles
  - ◊ utiliser les techniques vues pour les serveurs (l'accès aux données est géré par un serveur)
  - ◊ ou des techniques spécifiques selon les besoins en termes de cohérence (stricte/relâchée)
- Techniques matérielles
  - ◊ Disques RAID
  - ◊ SAN (Storage Area Network) avec duplication

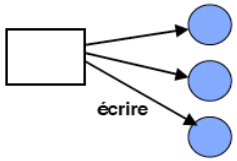
1)Techniques logicielles

Solutions

- Un serveur par jeu de données, avec utilisation d'une technique de disponibilité pour serveurs (primaire-secours ou duplication active)
- Ou bien : technique propre aux données dupliquées
  - ◊ deux opérations : lecture et écriture
  - ◊ technique élémentaire : N copies, lire un, écrire tous



Hypothèse :
toutes les copies sont identiques



Pour assurer que toutes les copies restent identiques : la diffusion des écritures doit être totalement ordonnée si les mises à jour ne commutent pas

  - ◊ inconvénient : écriture impossible si 1 copie en panne

Vote majoritaire

Principe

- Une opération (lire/écrire) est permise seulement si une majorité de copies est disponible
- Chaque copie a un numéro de version (incrémenté à l'écriture)

Exemple : 5 copies

- pour écrire, il faut au moins 3 copies disponibles
- pour lire,
  - ◊ on cherche 3 copies ayant le même numéro de version (nécessairement le plus récent)
  - ◊ ou on cherche 3 copies disponibles, et on est certain que l'une au moins est une copie à jour de la dernière modification.
    Les estampilles permettent de connaître la version la plus récente

Quorums

Principe

Autoriser une opération (écriture ou lecture) seulement si un quorum est atteint (nombre de copies disponibles)

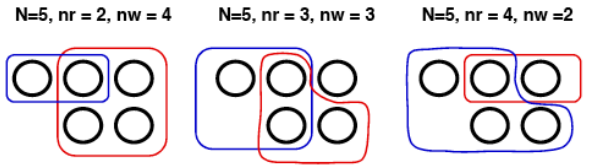
Les quorums en lecture et écriture sont définis en fonction des propriétés souhaitées

Exemple

N = nombre de copies

nr = quorum de lecture, nw = quorum d!écriture,

nr + nw > N garantit que l'on pourra toujours accéder en lecture à la version la plus récente



Vote pondéré

But

Utilisation de quorums/votes pour traiter les partitions du réseau

Idée

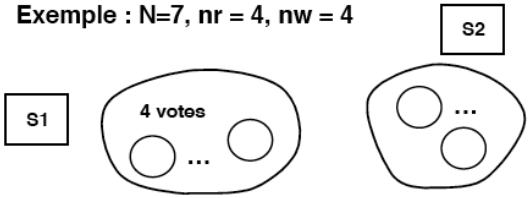
Donner des poids différents aux copies selon leur importance présumée (raisons d'accessibilité, administration, etc.)

N = nombre de votes disponibles (et non plus nombre de copies)

nr = quorum de lecture, nw = quorum d'écriture

nr + nw > N et 2nw > N

Exemple : N=7, nr = 4, nw = 4



Si S1 écrit dans la partition 1, on est sûr que S2 ne peut pas écrire ou lire dans la partition 2 (elle ne peut avoir plus de 3 votes)

2)Techniques matérielles

Disques RAID (Redundant Array of Inexpensive/Independent Disks)

Motivations

- Augmenter les performances (bande passante) par l'accès parallèle
- Augmenter la disponibilité par la redondance

Historique

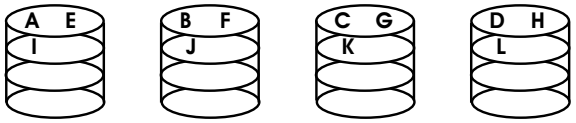
- Origine : Berkeley, 1987
- Nombreux produits commerciaux (HP AutoRaid, IBM, StorageTek, ...)

Classification

- Entrelacement par bancs (Striping) : RAID 0
- Miroir : RAID 1
- Accès parallèles synchronisés : RAID 2 et 3
- Accès indépendants : RAID 4 et 5
- Double redondance : RAID 6
- Extensions diverses (ex : allocation dynamique)

RAID 0 (Agrégation par bancs)

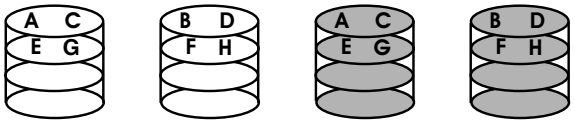
Entrelacement séquentiel sur l'ensemble des disques



- Accès parallèle possible à des enregistrements indépendants
- Pas de protection des données contre les pannes → panne d'un disque ⇒ perte de données
- Usage : calcul scientifique (performances d'accès = facteur dominant)

RAID 1(Miroir)

Entrelacement séquentiel sur la moitié des disques

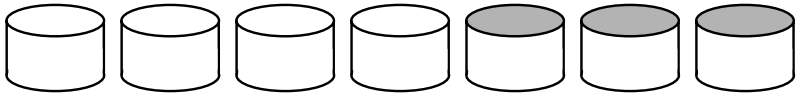


Disques miroir

- Accès parallèle possible à des enregistrements indépendants
- Redondance par duplication intégrale (miroir) → coût de la redondance : 50% de la capacité
- Usage : SGBD avec exigence forte de disponibilité et peu de contraintes de capacité

RAID 2 (Agrégation par bancs, redondance par code de Hamming)

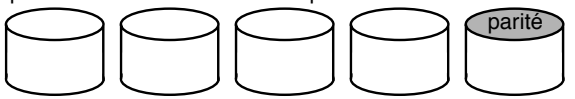
Entrelacement séquentiel à grain fin sur l'ensemble des disques



- Accès *synchrone* à l'*ensemble des disques*
- Redondance par code correcteur d'erreur (Hamming)
  - ◊ la capacité additionnelle requise croit comme log (nb. disques)
  - ◊ exemple : 4 (→ 3), 10 (→ 4)
- Usage : applications scientifiques (accès séquentiel, grand volume). Peu utilisé.

RAID 3

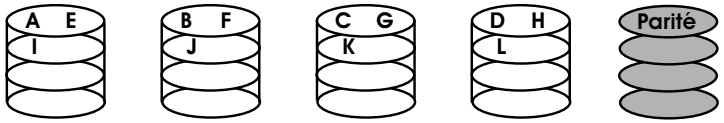
Entrelacement séquentiel sur l'ensemble des disques



- Accès synchronisés à l'ensemble des disques (redondance par octets)
- Redondance : informations de parité sur disque unique
  Reconstruire le contenu d'un disque défaillant requiert l'accès à tous les disques valides.
- Tolère la défaillance d'un disque
- Usage : transfert séquentiel de grands volumes de données. Amélioration : RAID 4.

RAID 4

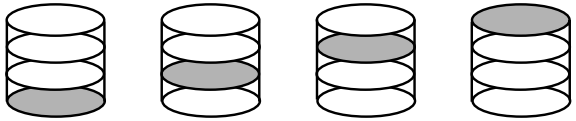
Entrelacement séquentiel sur l'ensemble des disques, par blocs



- Accès non synchronisés (accès indépendants possibles en parallèle) (redondance par blocs)
- Redondance : informations de parité sur disque unique
- Limitation :
  coût du maintien de la parité si mises à jour fréquentes - surcharge du disque de parité
- Amélioration : RAID 5

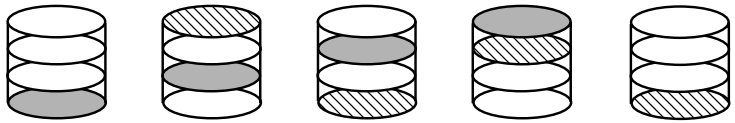
RAID 5 (Agrégation par bancs, avec parité répartie)

Entrelacement séquentiel sur l'ensemble des disques, par blocs



- Accès non synchronisés (accès parallèle possible)
- Redondance : informations de parité réparties sur tous les disques
  (élimine la surcharge du disque unique de parité)
- Efficace pour les lectures (toutes tailles) et les grandes écritures
- Peu efficace pour les petites écritures, et les écritures concurrentes sur un même banc

RAID 6



- Amélioration de RAID 5, tolère la défaillance de *deux* disques
- Caractéristiques : comme RAID 5, mais inefficacité accrue en écriture (mise à jour parité)

Conclusions sur RAID

- La plupart des produits utilisent RAID 0, RAID 1, ou RAID 5
- Le réglage d'un RAID est un processus délicat
  - ◊ nombreux paramètres de réglage (taille des bancs, ...)
  - ◊ sensibilité aux variations des paramètres de réglage
  - ◊ sensibilité aux variations de charge
  - ◊ coût de la reconfiguration en cas de changement
- Problèmes de performances dans certaines conditions
  - ◊ pour les petites écritures
  - ◊ en mode dégradé
- Solution encore relativement coûteuse
  - ◊ circuits spécialisés pour calcul de parité/synchronisation des accès
    → coût et point de défaillance unique
    difficulté d'accès au contenu en cas de panne du contrôleur)
  - ◊ solutions (semi) logicielles

    plus souple/robuste, mais consommation de ressources pouvant être importante,
    selon le degré de redondance (RAID 1 en particulier)