Systèmes et algorithmes répartis

Groupes, diffusion, tolérance aux fautes

Philippe Queinnec, Philippe Mauran

ENSEEIHT
Département Sciences du Numérique

24 octobre 2024

Plan

Contenu de cette partie

1. Abstraction de la communication entre plusieurs sites, en présence de défaillances
   - Ensemble de sites → groupe
   - Diffusion : définitions, propriétés
   - Prise en compte des défaillances : fiabilité, uniformité, atomicité
   - Mise en œuvre
   - Exemple : JavaGroups
2. Tolérance aux fautes

Situation

Difficultés majeures de la répartition

- asynchronisme
- absence d'état global
- défaillances partielles

Nécessité de services/abstractions pour réduire/encadrer l'incertitude liée aux défaillances :

- en termes de contrôle $\rightarrow$ décision partagée
  $\rightarrow$ consensus, détecteur de défaillance
- en termes de propagation de l'information
  $\rightarrow$ protocoles de groupes, diffusion atomique

Plan

Groupe de processus

Permettre de transmettre une information de manière homogène à un ensemble (groupe) de sites désigné

Opérations sur les groupes de processus :

- appartenance (*group membership*)
  - lister les membres du groupe (vue du groupe)
  - évolution du groupe : ajout/retrait de membres
- diffusion : communication d'ensemble vers les membres du groupe

Utilisation :

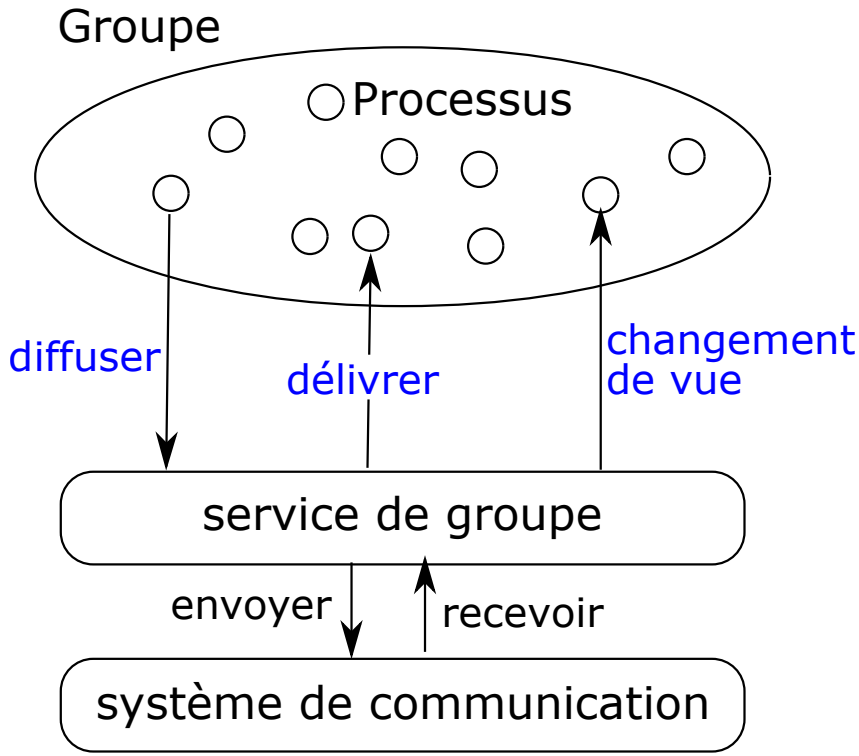
- travail coopératif, partage d'information : chat, simulation répartie
- tolérance aux fautes, équilibrage de charge : gestion d'un ensemble de services redondants, ou de données dupliquées

1. *Reliable Distributed Computing with the Isis Toolkit*, Kenneth P. Birman and Robbert van Renesse. IEEE Computer Society Press, 1994.

Service de groupes : interface

- diffuser(p,m) : le processus p diffuse le message m au groupe
- délivrer(p,m) : le message m est délivré à p
- changement-de-vue(p,id,V) : une nouvelle vue V (= ensemble de processus) identifiée par id est délivrée au processus p

La diffusion se fait toujours dans la vue courante



Protocoles de groupe : l'appartenance

Le service d'appartenance (*membership*) vise à fournir à tout instant à chaque membre du groupe une vue de la composition courante du groupe.

- Partition primaire : la suite des vues fournies aux membres est totalement ordonnée. Il n'y a pas de partition, ou de manière équivalente, un unique primaire.
- Partition multiple : la suite des vues fournies aux membres est partiellement ordonnées. Il peut y avoir simultanément plusieurs vues disjointes.

Diffusion

Diffusion générale (*broadcast*)

- destinataires = processus d'un seul ensemble (implicite)
- l'émetteur est aussi destinataire.

  Exemples
  - les membres d'un groupe unique, vus de l'intérieur du groupe ;
  - « tous » les processus joignables

Diffusion de groupe (diffusion sélective, *multicast*)

- les destinataires sont les membres d'un (ou plusieurs) groupe(s) désignés(s) explicitement.
- L'émetteur peut être extérieur au(x) groupe(s)

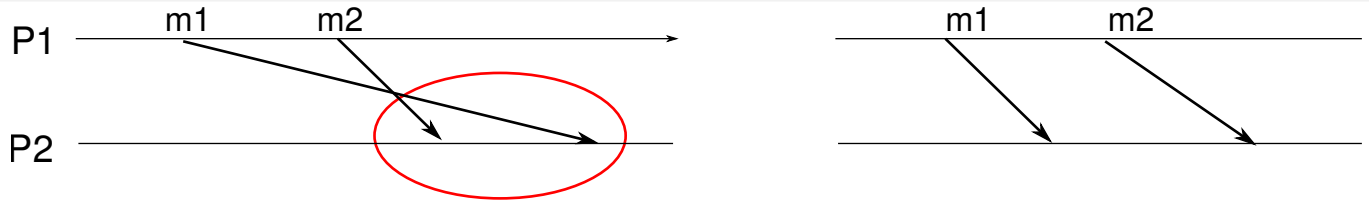
Interface

```
broadcast (émetteur, message)
multicast (émetteur, message, groupes destinataires)
deliver(récepteur, message)
```

Propriétés relatives à l'ordre d'émission

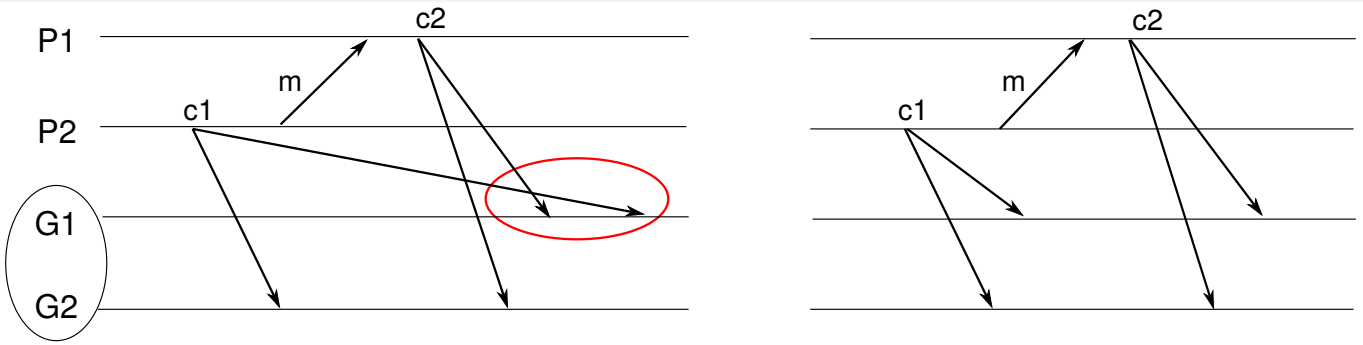
Diffusion FIFO

Deux messages d'un même émetteur sont délivrés (à chaque membre du groupe) dans l'ordre de leur envoi.



Diffusion causale

Si deux diffusions sont causalement liées, l'ordre de délivrance des messages respecte cet ordre causal.



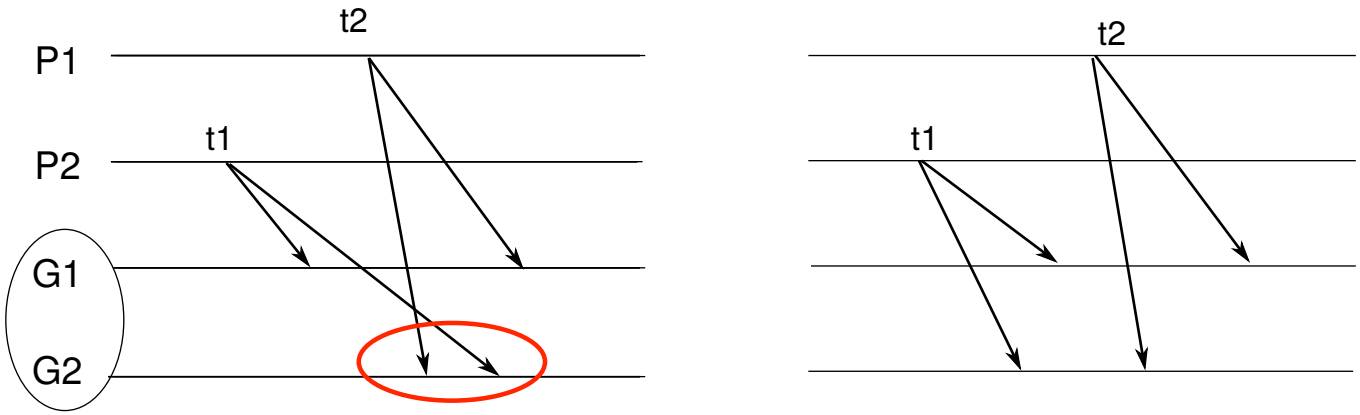
Propriétés indépendantes de l'ordre d'émission

Diffusion fiable

Un message est délivré à tous les destinataires corrects ou à aucun.

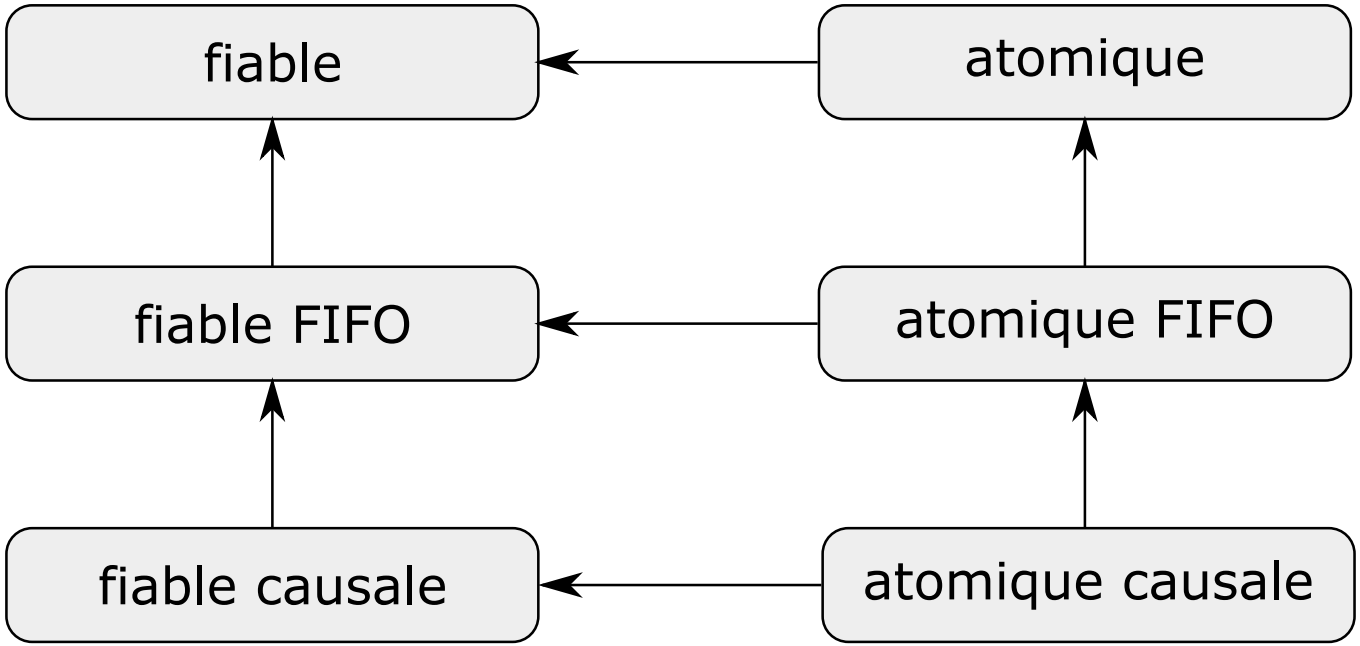
Diffusion totalement ordonnée (*diffusion atomique*)

La diffusion est fiable et les messages sont délivrés dans le même ordre sur tous leurs destinataires.



L'ordre total peut (ou non) être compatible avec la causalité

Synopsis

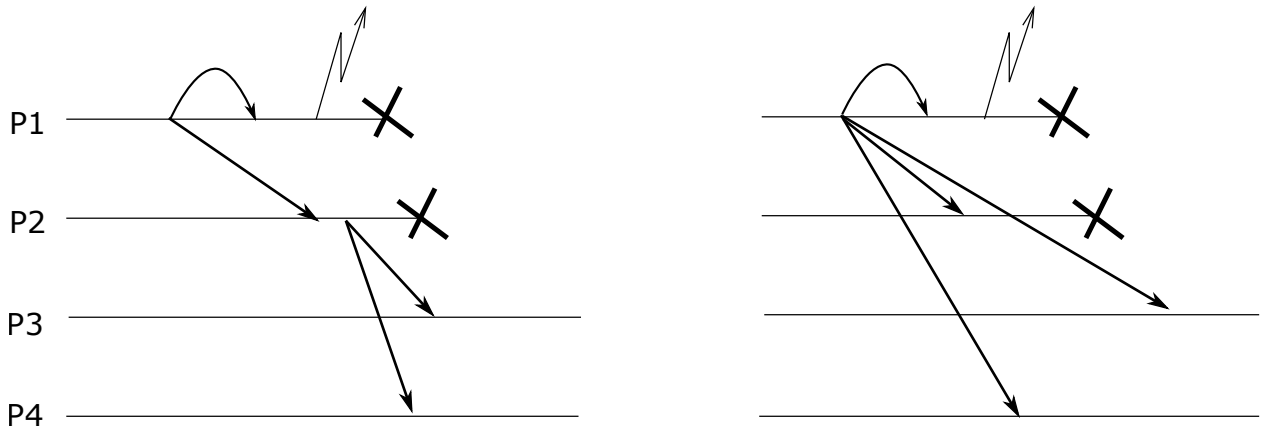


- $\rightarrow \approx$ implication logique/raffinage
- propriétés combinables avec
  - des contraintes de temps (délai maximum pour la délivrance)
    - $\Rightarrow$ hypothèses de synchronisme sur le service de communication
  - une extension à tous les sites (corrects ou fautifs) : uniformité

Uniformité

Diffusion fiable uniforme

Si un message est délivré à un processus (correct ou défaillant), il sera délivré à tous les processus corrects



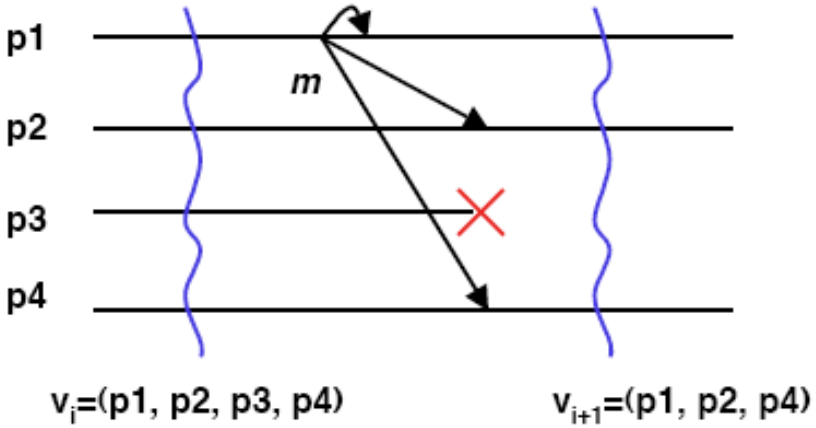
Non uniforme (mais fiable) Uniforme

Nécessaire si un processus défaillant peut (avant sa défaillance) provoquer des actions irréversibles, notamment vis-à-vis de son environnement (effets de bord).

Diffusion dans la vue

Vue = liste des processus valides actuellement dans le groupe

La diffusion fiable au sein de la vue assure que les processus présumés corrects et effectivement corrects reçoivent le même ensemble de messages.



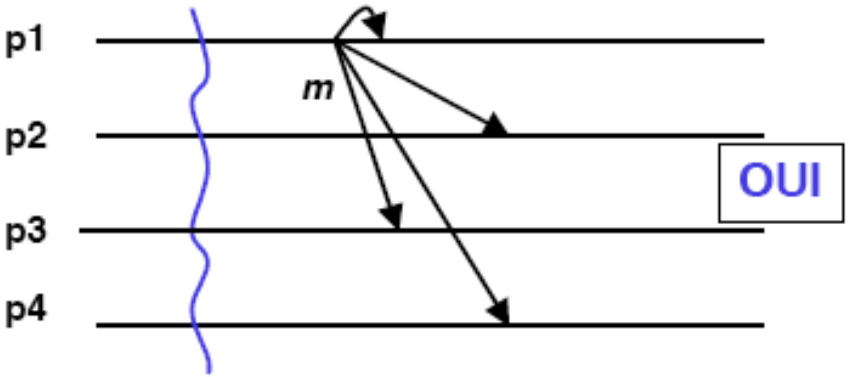
Soient

- m le message diffusée dans v_i
- $P = v_i \cap v_{i+1}$

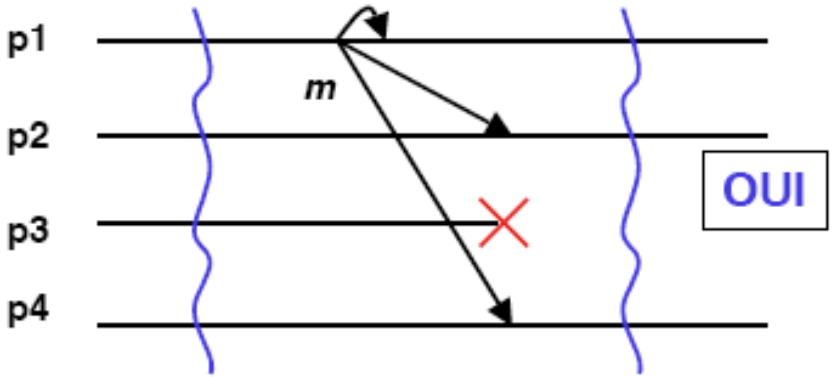
Alors m doit être délivré avant l'installation de v_{i+1}

- soit à tous les membres de P,
- soit à aucun membre de P.

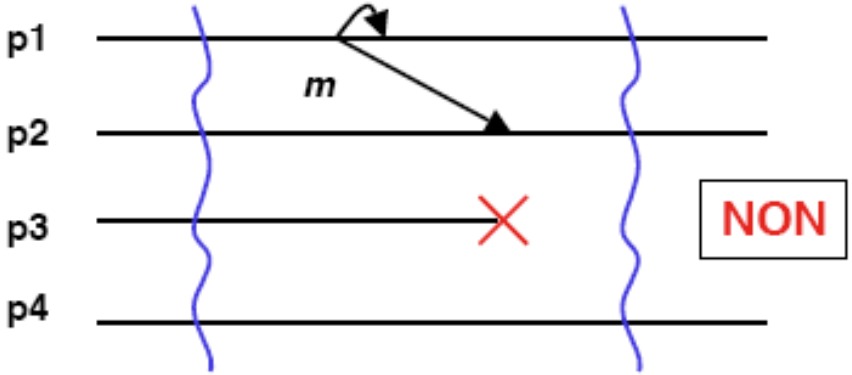
Vues synchrones



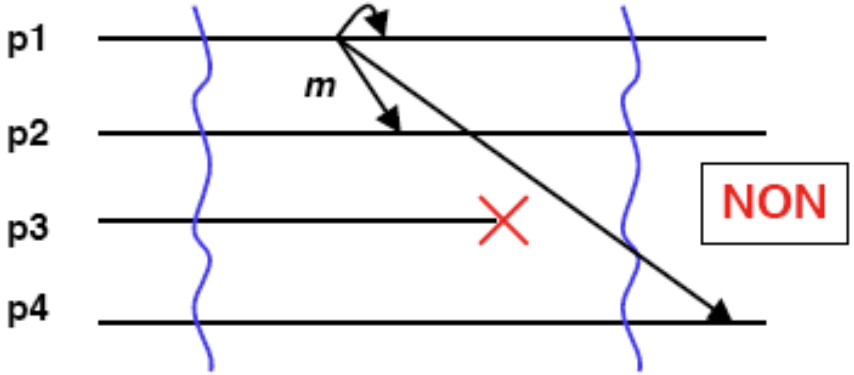
v_i=(p1, p2, p3, p4)



v_i=(p1, p2, p3, p4) v_{i+1}=(p1, p2, p4)



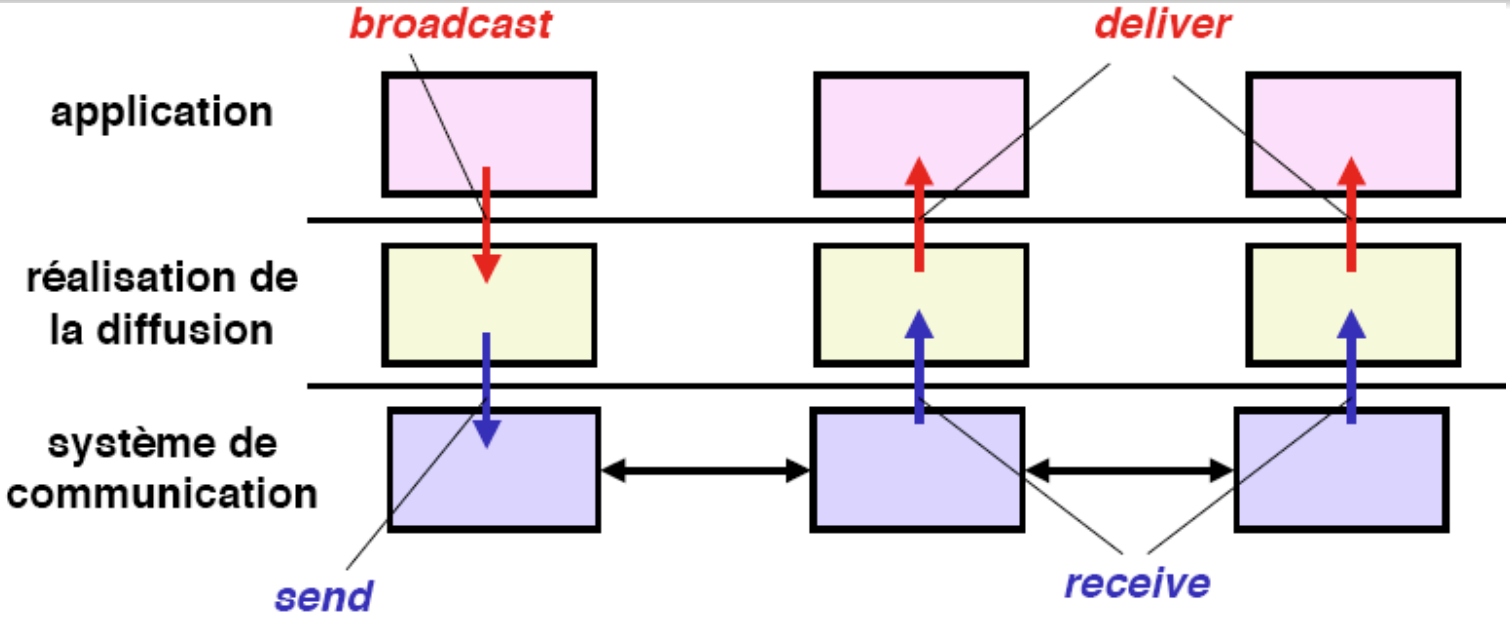
v_i=(p1, p2, p3, p4) v_{i+1}=(p1, p2, p4)



v_i=(p1, p2, p3, p4) v_{i+1}=(p1, p2, p4)

Plan

Réalisation de la diffusion (propriétés liées à l'ordre d'émission)



- Situation : la diffusion est réalisée au-dessus d'un service de communication permettant l'envoi et la réception de messages (primitives send et receive).
- Principe : distinguer la réception (au niveau du service de communication) et la délivrance (à l'application) :
  les messages reçus sont mis en attente, jusqu'à ce que les propriétés de diffusion souhaitées soient vérifiées.

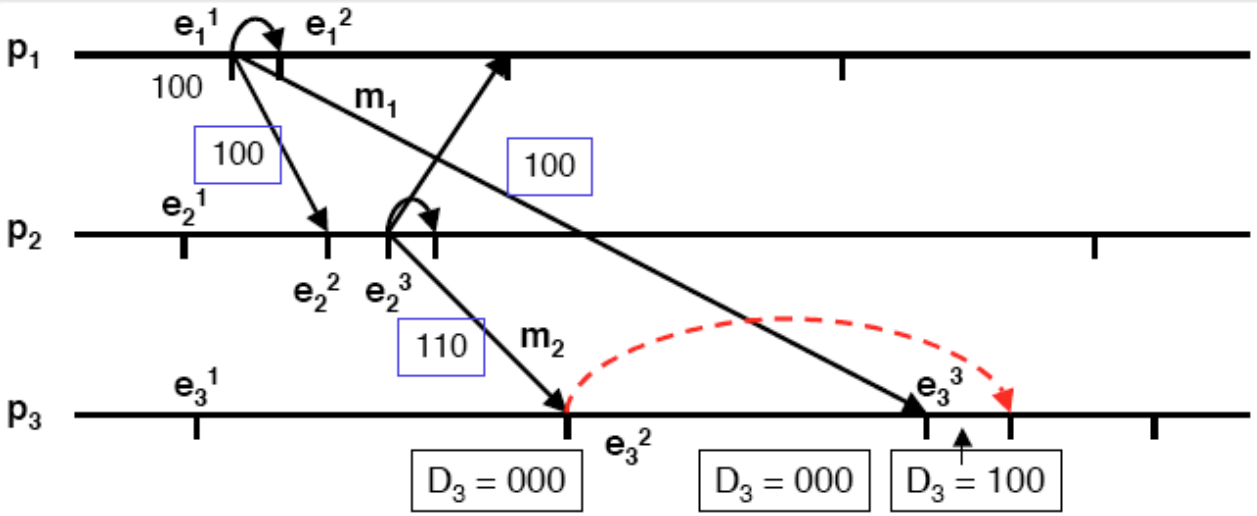
Exemples (rappels)

Diffusion FIFO

- messages estampillés dans l'ordre croissant par chaque émetteur
- un message estampillé ⟨id_émetteur, num⟩ n'est délivré qu'une fois délivré le message estampillé ⟨id_émetteur, num-1⟩.

Diffusion causale

- diffusions datées sur chaque site par des horloges vectorielles
- messages estampillés par la date de diffusion
- message délivré si son passé causal est inclus dans celui du récepteur



Résultat d'impossibilité

Hypothèses

- sites/processus sujets à défaillance d'arrêt
- service de communication
  - fiable : tout message finit par arriver, intact, si l'émetteur et le récepteur restent connectés au service (non défaillants)
  - asynchrone (délai de transmission fini non borné)

Alors

- La diffusion fiable (uniforme) est réalisable simplement.
- La diffusion atomique n'est pas réalisable.
  (résultat analogue pour le service d'appartenance : impossibilité de fournir à chaque membre du groupe une même suite de vues totalement ordonnée)

1. *Understanding the limitations of causally and totally ordered communication*, D. Cheriton, D. Skeen. SOSP 1993.
2. *On the Impossibility of Group Membership*, T.D. Chandra, V. Hadzilacos, S. Toueg, B. Charron-Bost. PODC 1996.

Spécification de la diffusion fiable *[uniforme]*

- Intégrité : Quel que soit le message m,
  - il est délivré au plus une fois à tout processus correct *[ou fautif]*,
  - et seulement s'il a été diffusé par un processus
- Accord : si un processus correct *[ou fautif]* délivre un message m, tous les processus corrects délivrent m (au bout d'un temps fini)
- Terminaison : si un processus correct diffuse un message m, tous les processus corrects délivrent m (au bout d'un temps fini)

[entre crochets : version uniforme]

Réalisation de la diffusion fiable (rappel)

Réalisation de broadcast(p,m) (p = processus émetteur de m)

pour tous les voisins de p (et p) **faire** send($\langle$m, p$\rangle$) **fin pour**

Contrôle de deliver(m)

Réalisation de receive($\langle$m, sender(m)$\rangle$) par le processus q :

si q n'a pas déjà exécuté deliver(m) **alors**
  si sender(m) $\neq$ q **alors**
    pour tous les voisins de q **faire** send($\langle$m, sender(m)$\rangle$) **fin pour**
  fin si
  deliver(m)
fin si

Tout site délivrant un message l'a auparavant envoyé à ses voisins

Réalisation de la diffusion fiable uniforme

Réalisation de broadcast(p,m) (p = processus émetteur de m)

pour tous les sites **faire**
  send(⟨m, p⟩)
fin pour

Contrôle de deliver(m)

Réalisation 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 sites **faire**
      send(⟨m, sender(m)⟩)
    fin pour
  fin si
  deliver(m)
fin si

Impossibilité de la diffusion atomique

Réduction

Avec une communication asynchrone fiable et des sites sujets à défaillance d'arrêt, la diffusion atomique se réduit au consensus.

impossibilité de la diffusion atomique ≡ impossibilité du consensus

i) Réaliser le consensus avec un algo. de diffusion atomique
- Chaque processus diffuse atomiquement sa valeur proposée à tous les membres du groupe.
- Tous les processus corrects reçoivent le même ensemble de valeurs dans le même ordre.
- Ils décident la première valeur, la même pour tous.

ii) Réaliser la diffusion atomique avec un algo. de consensus
- Les messages sont transmis par diffusion fiable.
- Algorithme par tours : à chaque tour, chaque site propose l'ensemble des messages reçus non encore délivrés.
- Le consensus permet de déterminer un ordre commun pour les messages à délivrer dans le tour courant (s'il y en a).

Réalisation de la diffusion atomique

En pratique

Les algorithmes ≪ pratiques ≫ de réalisation de la diffusion atomique doivent donc relâcher des contraintes :

- introduire du synchronisme (utiliser un délai de garde ou supposer un détecteur de défaillances)
- interdire les arrêts définitifs (supposer que certains processus défaillants peuvent être rétablis)

Construire de l'ordre global

- utiliser un service produisant une suite croissante (séquenceur)
- construire l'ordre à l'émission
- construire l'ordre 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.

- Le séquenceur peut être fixe ou circulant (jeton)
- Difficulté : résister à la défaillance du séquenceur
  $\rightarrow$ redondance (détection (scrutation périodique), élection, reprise)
- Utilisé dans le protocole JGroups

Ordre défini à l'émission

Estampilles déterminées au moment de l'émission

- Exclusion mutuelle pour l'émission (ex : jeton)
  $\rightarrow$ ordre de passage en exclusion mutuelle
- $\langle$Horloges logiques (Lamport), id site$\rangle$
- ou ordre préétabli

- Principe de l'ordre construit : Lorsqu'un site connaît l'estampille du prochain message proposé par chacun des sites, il peut déterminer le prochain message à délivrer : c'est celui dont l'estampille est la plus petite.
- Difficulté ordre construit : vivacité
  - $\rightarrow$ canaux FIFO
  - $\rightarrow$ envoi périodique de messages vides.

Ordre fixé à la réception (ABCAST d'Isis)

- Chaque message m est estampillé provisoirement par son heure logique de réception (horloges de Lamport).
- Les différents récepteurs se communiquent leurs estampilles provisoires (l'émetteur peut jouer le rôle de collecteur)
- Quand toutes sont connues, on attribue définitivement à m la plus grande : le message a la même estampille définitive sur tous les sites
- Les messages sont délivrés dans l'ordre des estampilles définitives

Quand un message d'estampille définitive e est délivré, e minore toutes les estampilles définitives (fixées ou à venir) de messages non encore délivrés sur le site :

- l'estampille définitive majore l'estampille provisoire
- avec des canaux FIFO, les estampilles provisoires à venir majorent les estampilles provisoires déjà reçues
  $\rightarrow$ les estampilles définitives à venir majorent les estampilles définitives déjà fixées

Difficulté : défaillances $\rightarrow$ envois redondants + délais de garde

Plan

Etude de cas : JGroups

Gestion de groupe, et services de diffusion élaborés :
diffusion fiable, ordonnée, causale, sélective...

Références

- site du projet JGroups (en anglais)
  `http://www.jgroups.org/`
- Fournit la documentation (API, manuels, tutoriels, exemples), et les liens de téléchargement
- le blog de l'auteur principal du projet
  `http://belaban.blogspot.com`
- Forum : `https://groups.google.com/g/jgroups-dev`

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(protocole.xml) / ch.close()
- rejoindre un groupe utilisant le canal : ch.connect("nom") le 1^{er} processus qui rejoint le groupe crée le groupe, s'il n'existe 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 s'abonne auprès du canal par ch.setReceiver(rcv),
  - où `rcv` implante l'interface Receiver avec une méthode receive(message), appelée à chaque délivrance.

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 accesseurs permettent de communiquer l'état d'un objet associé au groupe : membre.getState()/setState(état).
  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 ch.getState est utile en particulier lorsqu'un membre rejoint un groupe
  - 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.Receiver;
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 Chat implements Receiver {
    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, line.getBytes());
                channel.send(msg);
            } catch(Exception e) {}
        }
    }

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

Configuration : pile de protocoles

Exemple d'une pile

- UDP : protocole de transport
- PING : découverte de processus par IP multicast
- MERGE3 : fusion de sous-groupes après partition
- FD_SOCK : détecteur de défaillances basé sur les sockets
- FD_ALL : détecteur de défaillances avec message *are-you-alive*
- VERIFY_SUSPECT : confirmation de défaillances
- BARRIER : blocage des msg pour transfert d'état cohérent
- pbcast.NAKACK2 : diffusion fiable et fifo
- pbcast.STABLE : purge des messages reçus par tous
- pbcast.GMS : *group membership*
- UFC : *unicast flow control* (entre deux membres)
- MFC : *multicast flow control* (entre tous)
- FRAG2 : fragmentation des gros messages
- STATE_TRANSFER : transfert d'état à un nouveau membre

Bilan JGroups

- Interface simple
- Composition de protocoles simples, modulaires
  - transport (UDP, TCP)
  - découverte (ping...)
  - diffusion probabiliste
  - fiabilité, ordre FIFO, ordre total
  - appartenance, gestion de groupe
  - détecteurs de défaillance
  - transfert d'état
  - fragmentation
  - compression
  - sécurité
  - ...

  → outils/protocoles de base pour réaliser les protocoles de plus haut niveau (diffusion ordonnée, causale)
- Logiciel libre
  - communauté active, mais assez réduite

Plan

Faute → erreur → défaillance

Défaillance (*failure*)

Déviation par rapport à la spécification.
Manifestation d'une erreur vis-à-vis du service.

Erreur (*error*)

Partie de l'état du système entraînant la défaillance.
Manifestation d'une faute vis-à-vis du système.

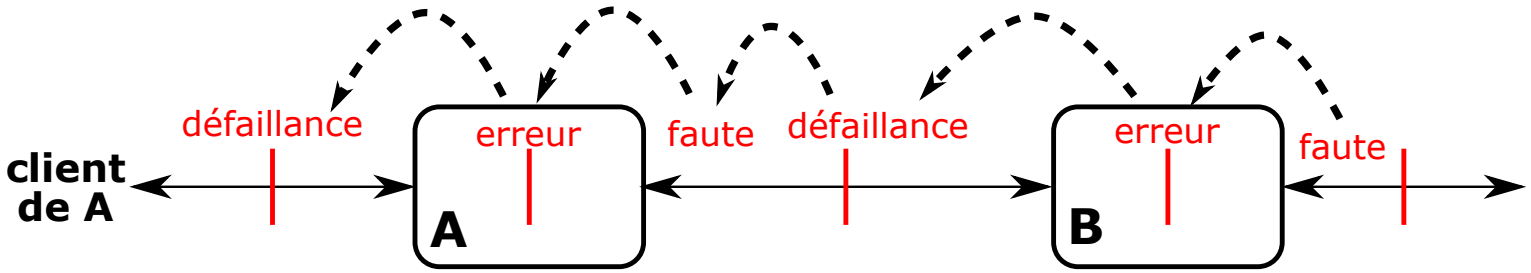
Note : une erreur est susceptible de causer une défaillance mais pas nécessairement (erreur latente, erreur compensée...)

Faute (*fault*)

Cause de l'erreur.

Ex : faute de programmation ⇒ division par 0 ⇒ arrêt du service
virus de la grippe ⇒ malade ⇒ absent en cours

Cascade/Propagation



- Le bon fonctionnement de A dépend du bon fonctionnement de B
- Une défaillance de B constitue une faute pour A
- Cette faute peut à son tour provoquer une erreur interne de A, causant une défaillance de A

Ex : virus de la grippe ⇒ malade ⇒ absent en cours
absent en cours ⇒ rien compris ⇒ UE non validée

Classification des défaillances

Un serveur doit rendre un service correctement, i.e. conformément à une spécification (valeur, transition, vitesse...).

- défaillance d'omission : le service ignore une requête
- défaillance temporelle : la réponse est fonctionnellement correcte mais pas au bon moment
  - défaillance temporelle d'avance : réponse trop tôt
  - défaillance temporelle de retard : réponse trop tard (défaillance de performance)
- défaillance de réponse : réaction inadaptée à la requête
  - défaillance de valeur : valeur renvoyée erronée
  - défaillance de transition : transition interne erronée
- défaillance arbitraire ou byzantine

Défaillance d'arrêt

Si, suite à une première omission, le serveur ignore toutes les requêtes jusqu'à ce qu'il soit redémarré, on parle de panne franche, défaillance d'arrêt, *crash failure* ou *fail stop*.

Redémarrage

- *amnesia-crash* : état initial indépendant de l'état avant crash
- *pause-crash* : état identique à l'état avant crash
- *halting-crash* : pas de redémarrage (crash définitif)

Défaillance d'arrêt = cas le plus simple, auquel on essaie de se ramener (arrêt forcé sur tout type de défaillance)

Note : un client ne peut pas distinguer entre un serveur « planté », un serveur lent ou une défaillance du réseau.

Système fiable, sûr et disponible ?

Évitement des fautes

Empêcher les fautes de se produire

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

Tolérance aux fautes

Assurer le service malgré l'existence de fautes → redondance

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

La tolérance aux fautes est indispensable :

- l'évitement des fautes est coûteux
- l'occurrence de fautes est inévitable

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 (échéances anormales, 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*)

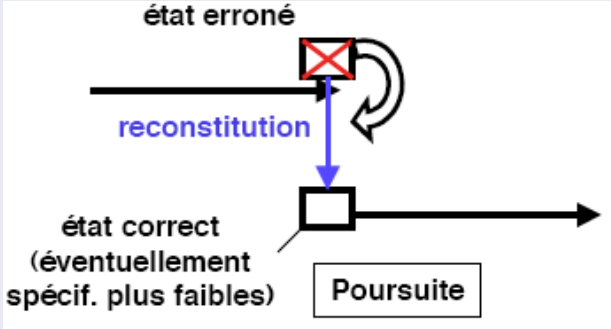
Le système a une redondance interne suffisante pour corriger l'erreur de manière transparente pour les utilisateurs

- composants dupliqués
- traitements dupliqués

Récupération

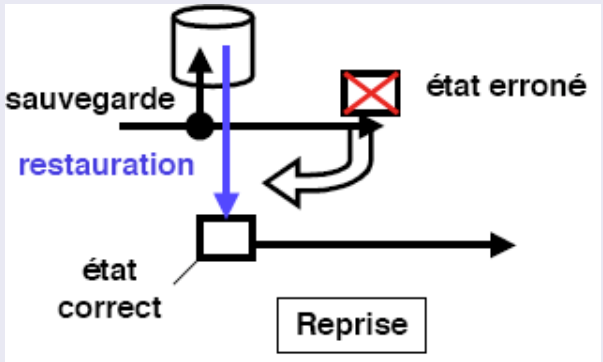
Poursuite (*forward recovery*)

- reconstitution d'un état correct, à partir de l'état courant
- reconstitution souvent partielle → service dégradé
- spécifique à chaque application



Reprise (*backward recovery*)

- retour en arrière vers un état antérieur connu correct
- nécessite la sauvegarde d'états corrects
- technique générale



Points 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

- la sauvegarde et la restauration doivent être atomiques
- la construction des points de reprise doit elle-même être protégée contre les fautes

Reprise dans un système réparti

Principe

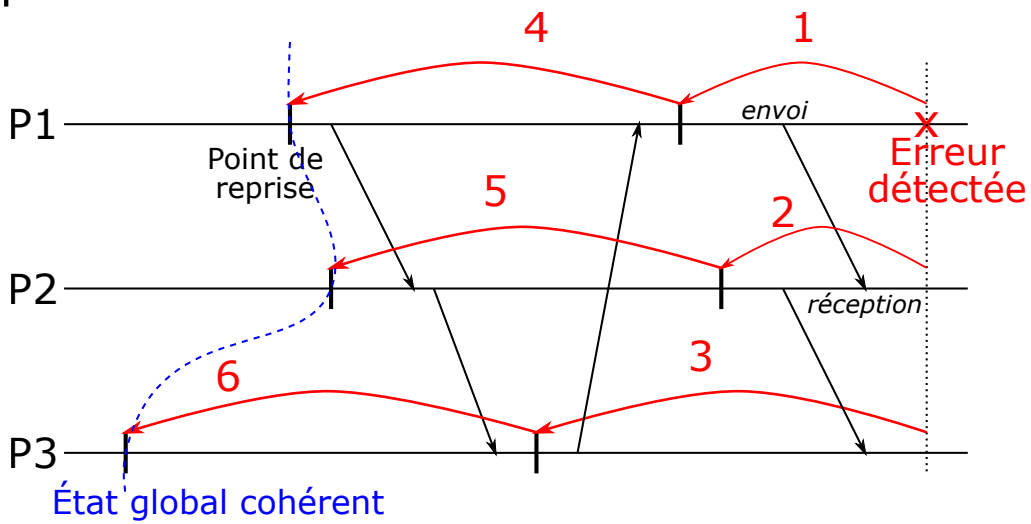
- Sauvegarde d'un état global
- Faute → restauration d'un état sauvegardé

Difficultés

- 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
- L'état restauré doit être cohérent
  - L'enregistrement de sauvegardes est local à chaque site
  - La cohérence est une propriété globale

Construction d'un état global cohérent

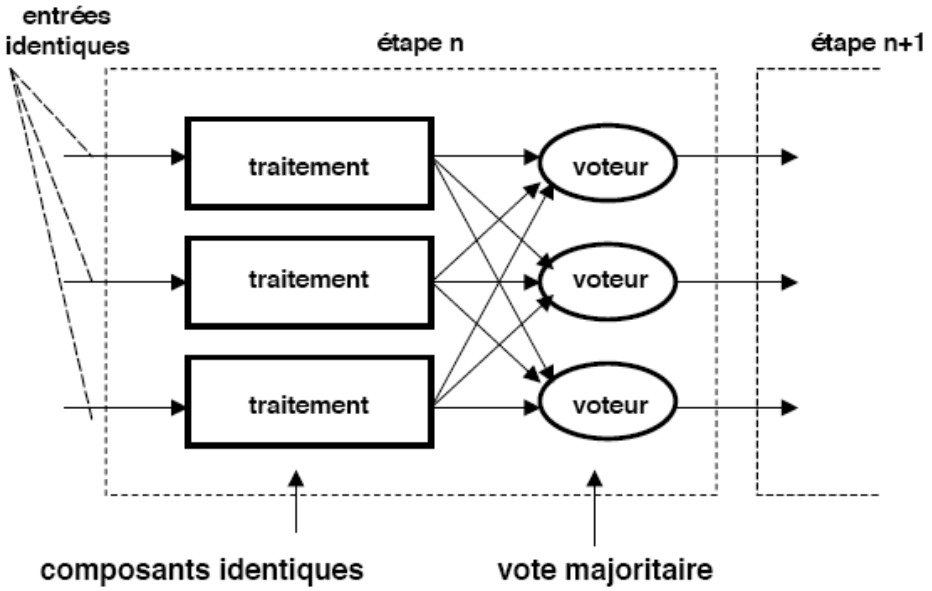
- Collecte synchronisée des états locaux à un même instant logique → algorithmes coûteux de prise de clichés (ex : Chandy-Lamport)
- Reconstruire un état cohérent à partir d'enregistrements locaux non coordonnés
  - Traitement réalisé hors ligne (coût réduit à l'exécution)
  - Pas d'interaction entre sites
  - Risque d'effet domino



Compensation par vote majoritaire

Traitements redondants pour aboutir à au moins un traitement correct

Exemple : architecture TMR (*Triple Modular Redundancy*)



(dessin : S. Krakowiak)

- Vote majoritaire
- Résiste à la défaillance d'un composant de traitement et/ou d'un voteur par étape
- Transparent, sans délai
- Coûteux

Service tolérant aux fautes

Redondance : N serveurs pour résister à $N - 1$ défaillances d'arrêt

Redondance froide (reprise)

- Points de reprise
- Serveurs de secours prêts à démarrer

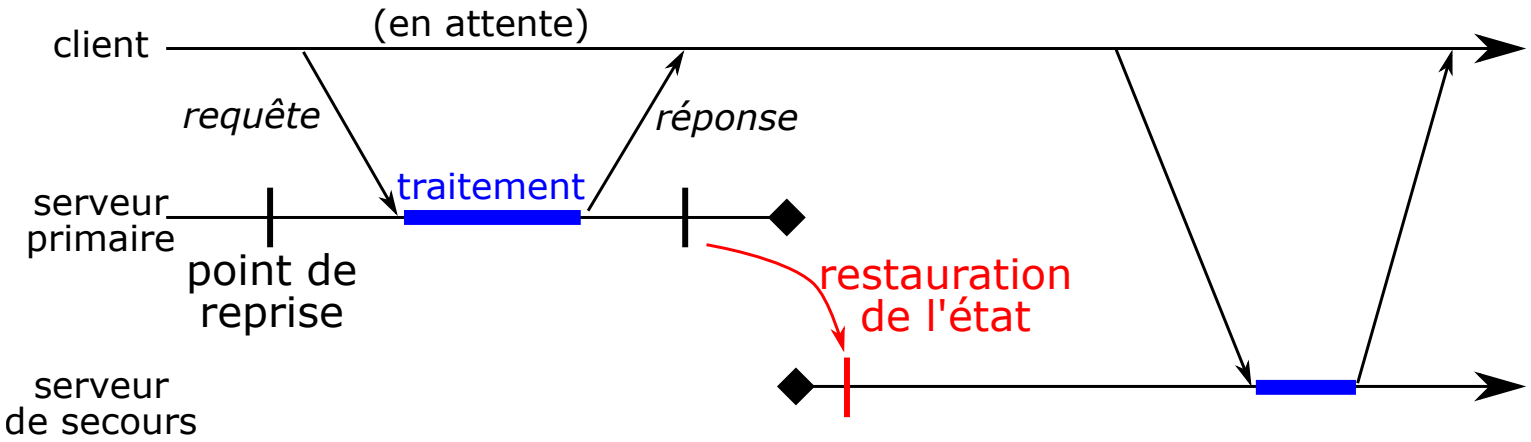
Redondance passive (récupération)

- Serveur de primaire + serveurs de secours
- Un seul serveur (le primaire) exécute les requêtes des clients
- La défaillance du primaire est visible des clients

Redondance active (compensation)

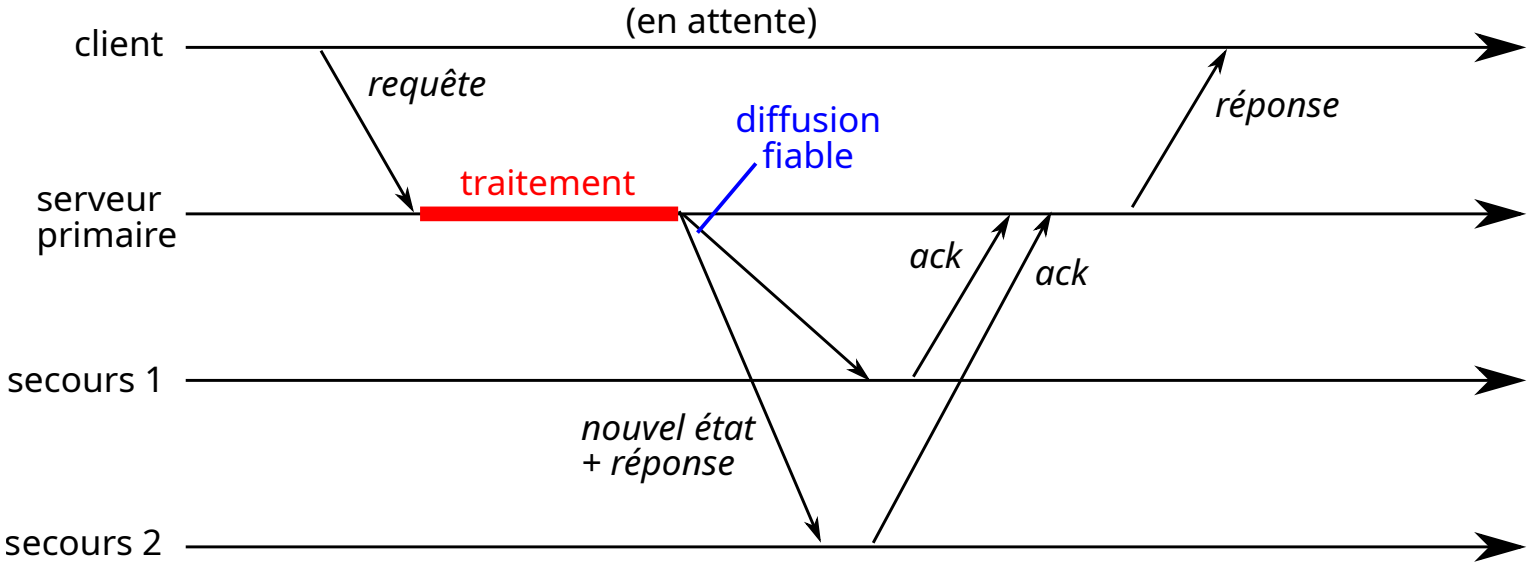
- N serveurs symétriques exécutant toutes les requêtes
- La défaillance d'un serveur est invisible aux clients (tant qu'il reste un serveur !)

Serveur primaire – secours froids



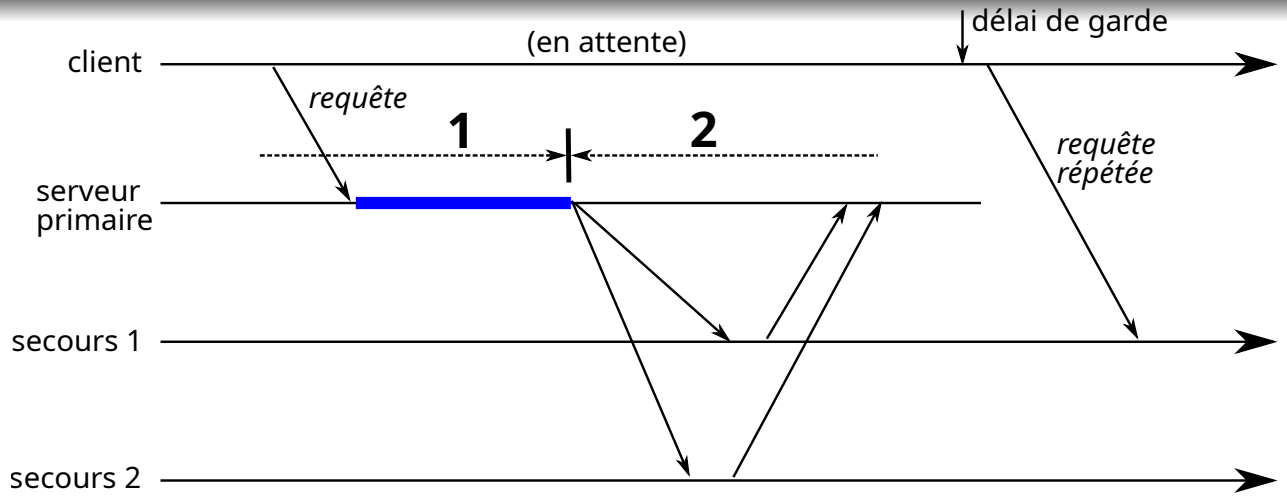
- Défaillance visible par le client
- Points de reprise : périodiquement, ou à chaque transition importante
- Simple mais latence de reprise
- Retour en arrière ⇒ requêtes traitées mais oubliées

Serveur primaire – serveurs de secours



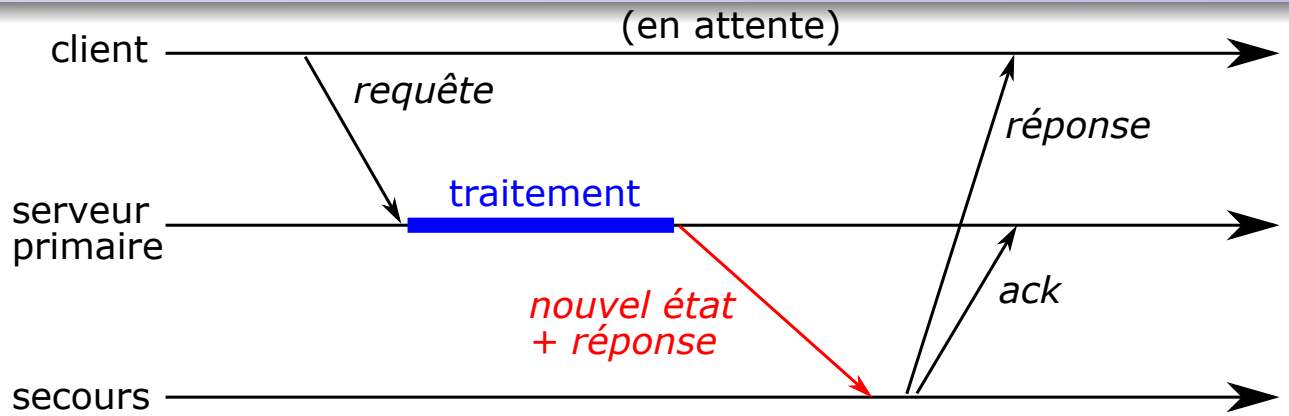
- Le primaire traite seul les requêtes et les traite séquentiellement → ordre global assuré sur les traitements
- Cohérence : quand le primaire répond, tous les serveurs non défaillants sont dans le même état
- Poursuite : tout serveur de secours peut remplacer le primaire en cas de défaillance

Défaillance du serveur primaire



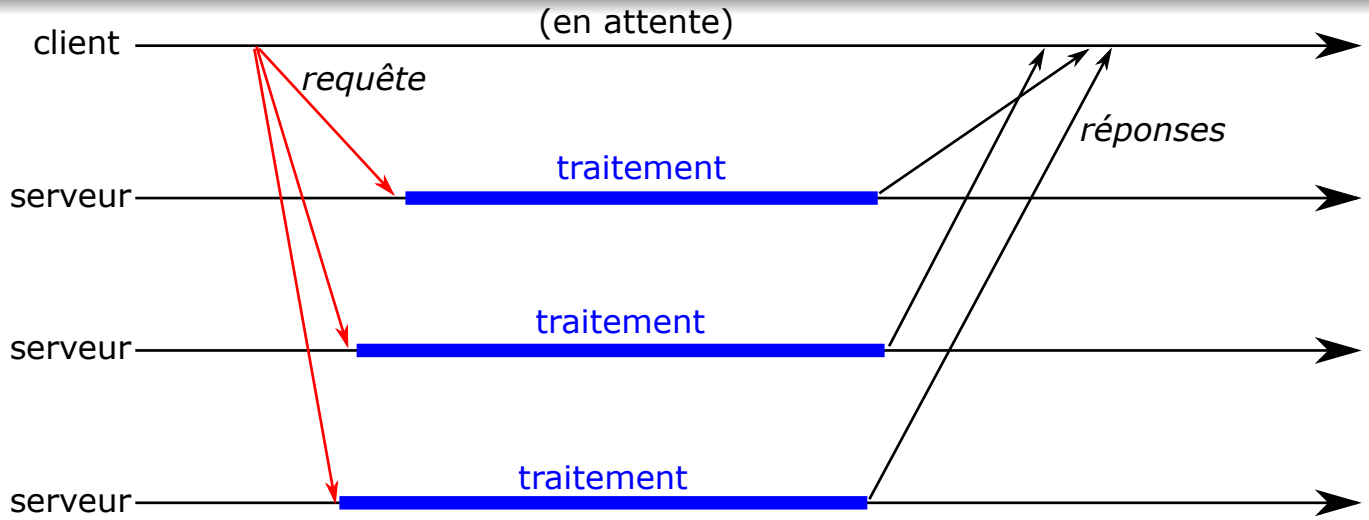
- Détection de la défaillance par le client et par les secours
  - Un des secours devient primaire (élection ou ordre défini à l'avance)
  - Le client renvoie sa requête au nouveau primaire
- Diffusion fiable ⇒ tous les serveurs sont à jour, ou aucun :
  - défaillance survenue en 1 (aucun serveur à jour)
    ⇒ comme une nouvelle requête
  - défaillance survenue en 2 (tous les serveurs à jour)
    ⇒ renvoie de la réponse déjà calculée
  - Les requêtes ont un identifiant unique, pour distinguer 1 et 2.

Cas particulier : un seul secours



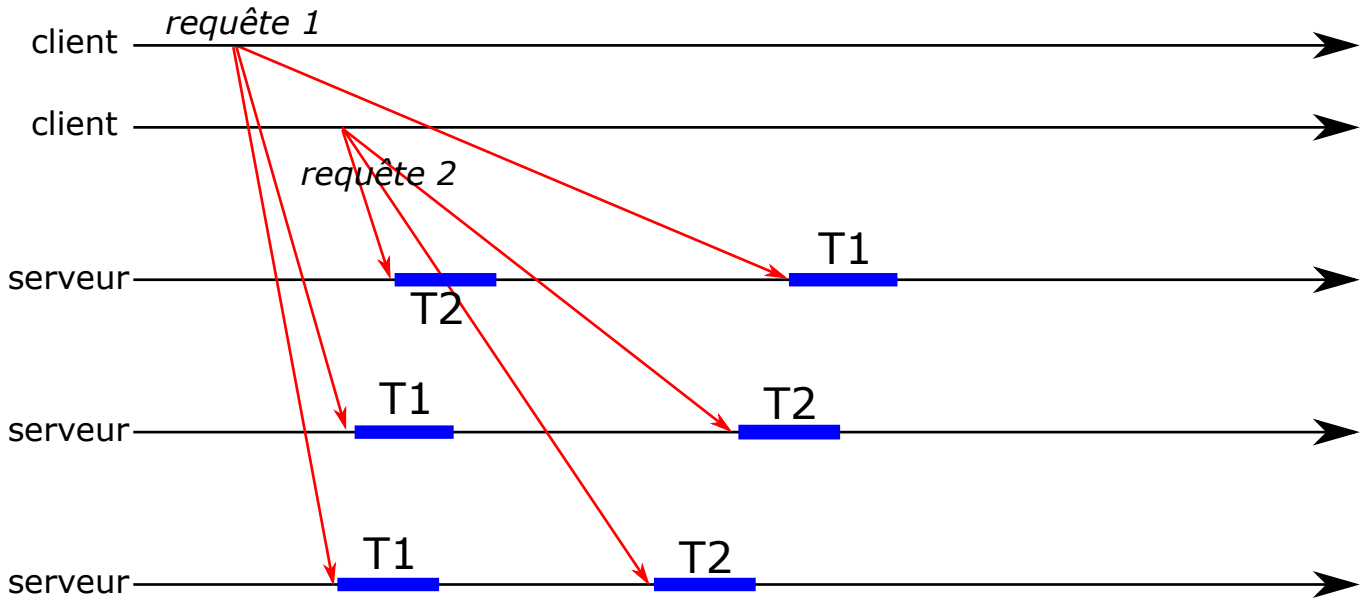
- Protocole plus simple (pas besoin de diffusion fiable)
- Réponse envoyée par le secours
- Défaillance du serveur primaire
  - Détection par le client qui bascule sur le secours (délai de garde)
  - Le secours devient primaire
  - Réinsertion de l'ancien primaire comme secours après réparation
- Défaillance du serveur de secours
  - Transparent pour le client
  - Réinsertion après réparation

Redondance active



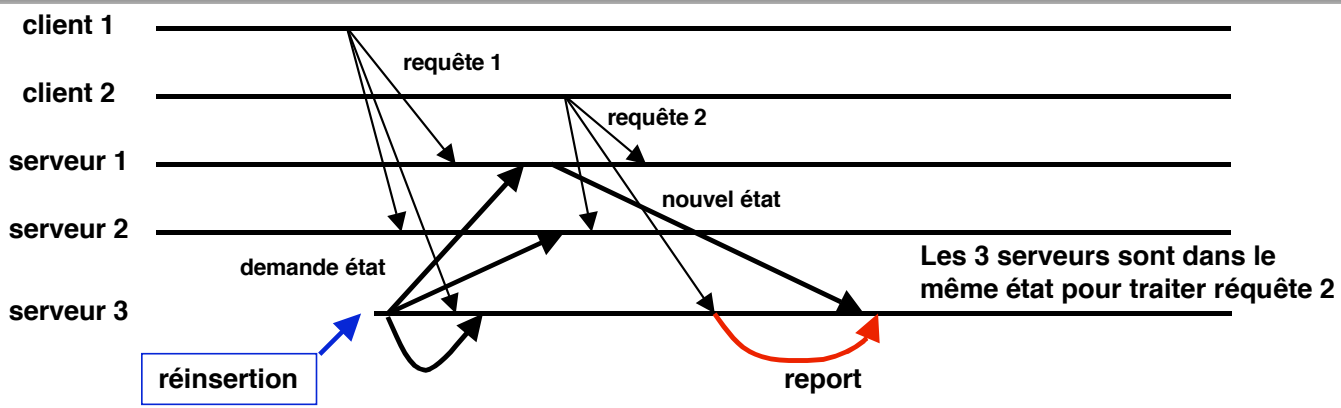
- Tous les serveurs sont équivalents, traitent les mêmes requêtes et exécutent le même traitement
- Le client choisit la première réponse (défaillance d'arrêt), ou la réponse majoritaire (défaillance arbitraire)
- Diffusion fiable et totalement ordonnée (atomique) :
  - Message délivré par tous les destinataires ou aucun
  - Deux messages différents sont délivrés dans le même ordre sur tous les destinataires communs

Nécessité de la diffusion atomique



Le serveur 1 traite la requête 2 avant la 1, alors que les serveurs 2 et 3 traitent 1 puis 2 $\Rightarrow$ état incohérent si les traitements modifient l'état des serveurs (traitements non commutatifs)

Réinsertion après défaillance



- 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 demande de réinsertion est délivrée permet de dater l'état transmis par rapport aux requêtes traitées :
  - Une requête (1) délivrée avant la demande de réinsertion est ignorée
  - Une requête (2) délivrée après la demande de réinsertion est retardée jusqu'à l'arrivée du nouvelle état

Modèle : réplication de machine à états

Machine à états

$M = (\mathcal{S}, \mathcal{I}, \mathcal{O}, \mathcal{T}, \omega, s_0)$ où

$\mathcal{S}, \mathcal{I}, \mathcal{O}, s_0$ = ensemble d'états / d'entrée / de sortie / état initial

$\mathcal{T}$ = fonction de transition $\mathcal{S} \to \mathcal{I} \to \mathcal{S}$

ω = fonction de sortie $\mathcal{S} \to \mathcal{I} \to \mathcal{O}$

$\mathcal{T}$ et ω sont déterministes.

Principe

- Répliquer la machine à états décrivant le service
- $\mathcal{I}$ = requêtes des clients, $\mathcal{O}$ = réponses aux clients
- Évolution asynchrone des réplicas (chacun à son rythme)
- Même ordre de traitement des requêtes + déterminisme $\Rightarrow$ même séquence de sorties pour tous les réplicas
- Défaillance détectée par divergence des états ou des sorties

1. *Using time instead of timeout for fault-tolerance in distributed systems*, Leslie Lamport, ACM TOPLAS, April 1984.

Comparaison redondance froide/passive/active

- Mécanismes nécessaires :
  - Diffusion fiable pour passive, fiable-atomique pour active
  - Coordination des départs/arrivées (notion de vue, à suivre)
- Ressources :
  - Serveur froid : éteint ; reprise longue
  - Serveur primaire : serveurs de secours non utilisés $\Rightarrow$ gratuits / utilisés pour autre chose ; reprise non immédiate
  - Redondance active : exécutions superflues ; reprise immédiate
- Transparence pour le client : uniquement en redondance active
- Sensibilité aux bugs logiciels : même état + même entrée $\Rightarrow$ même bug $\Rightarrow$ tous se plantent... (redondance froide moins sensible : retour en arrière)
- Système critique : combinaison des trois techniques compensation/poursuite/reprise : 2 serveurs actifs + 1 secours passif + 1 secours froid avec points de reprise

Conclusion

- Notion de groupe de processus, et de vues (évolution du groupe)
- Diffusion au sein de la même vue pour tous
- Diffusion atomique : fiable + totalement ordonnée
- Brique de base pour construire des serveurs tolérants aux fautes par réplication
- . . . mais impossibilité dans un système asynchrone avec défaillance d'arrêt