ARA
Algorithmique Répartie avancée
Master 2 - SAR

Luciana Arantes

Planning

- **Cours et TDs**
  - Protocole de Diffusion
  - Détecteur de Défaillance
  - Consensus Paxos
  - Checkpointing
  - Mémoire Partagée
  - Algorithmes Auto-stabilisants
  - Time Varying Graphs
- **TME + Devoirs**
  - PeerSim

Planning

- **Intervenants**
  - Luciana Arantes – Protocole diffusion, mémoire partagée
  - Swan Dubois – Graphes dynamiques
  - Jonathan Lejeune et Arnaud Favier – Peersim (TME, devoir)
  - Franck Petit – algorithmes auto-stabilisants
  - Pierre Sens – détecteurs défaillance, paxos
  - Julien Sopena – checkpointing
- **Evaluation**
  - Examen1 répartie (40%) + Examen réparti 2 (40%) + Devoir (20%)
    - Examen réparti 1 : Arantes et Sens
    - Examen réparti 2: Dubois, Petit et Sopena
    - Devoir: Lejeune

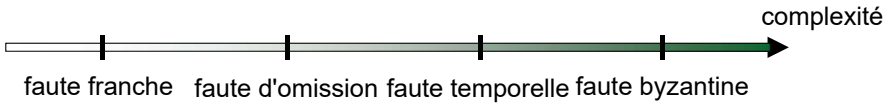
Rappels
Modèles de fautes et modèles temporels

Modèles de fautes

- **Origines des fautes**
  - fautes logicielles (de conception ou de programmation)
    - quasi-déterministes, même si parfois conditions déclenchantes rares
    - très difficiles à traiter à l'exécution : augmenter la couverture des tests
  - fautes matérielles (ou plus généralement système)
    - non déterministes, transitoires
    - *corrigées* par point de reprise ou *masquées* par réplication
  - piratage
    - affecte durablement un sous-ensemble de machines
    - masqué par réplication
- **Composants impactés**
  - Processus, processeurs, canaux de communication

Modèles de fautes

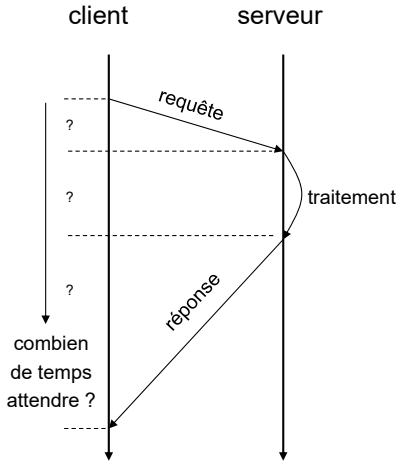
- **Classification des fautes**
  - ***faute franche*** : arrêt définitif du composant, qui ne répond ou ne transmet plus
  - ***faute d'omission*** : un résultat ou un message n'est transitoirement pas délivré
  - ***faute temporelle*** : un résultat ou un message est délivré trop tard ou trop tôt
  - ***faute byzantine*** : inclut tous les types de fautes, y compris le fait de délivrer un résultat ou un message erroné (intentionnellement ou non)



Modèles temporels

- **Constat**
  - vitesses processus différentes
  - délais de transmission variables
- **Problème**
  - ne pas attendre un résultat qui ne viendra pas (suite à une faute)
  - combien de temps attendre avant de reprendre ou déclarer l'échec ?
- **Démarche**
  - élaborer des modèles temporels

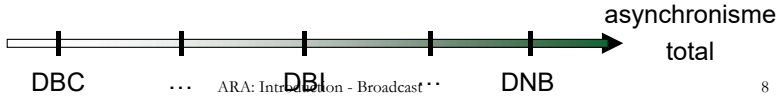
  dont on puisse tirer des propriétés



Modèles temporels (2)

Modèle temporel = hypothèses sur :
- délais de transmission des messages
- écart entre les vitesses des processus

système synchrone :	Modèle Délais/écarts Bornés Connus (DBC) - permet la détection parfaite de faute
système partiellement synchrone :	Modèle Délais/écarts Bornés Inconnus (DBI)
système asynchrone :	Modèle Délais/écarts Non Bornés (DNB)

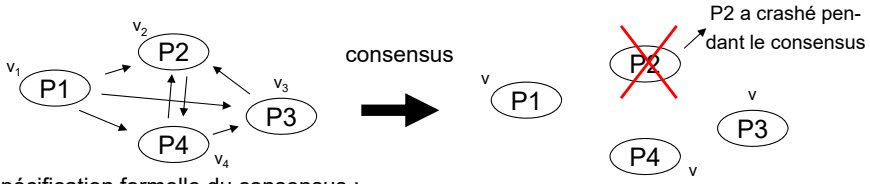


Modèles temporels (3)

Résultat fondamental :

Fischer, Lynch et Paterson 85 : le problème du consensus ne peut être résolu de façon déterministe dans un système asynchrone en présence de ne serait-ce qu'une faute franche.

Problème du consensus : N processus se concertent pour décider d'une valeur commune, chaque processus proposant sa valeur initiale v_i.



Spécification formelle du consensus :

- terminaison : tout processus correct finit par décider
- accord : deux processus ne peuvent décider différemment
- intégrité : un processus décide au plus une fois
- validité : si v est la valeur décidée, alors v est une des v_i

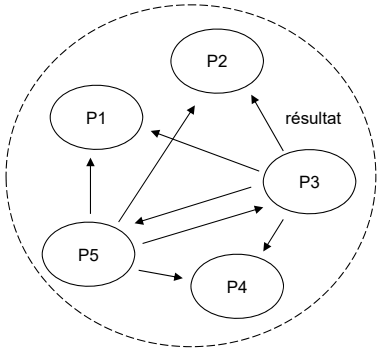
Notre modèle

- **Ensemble de processus séquentiels indépendants**
  - Chaque processus n'exécute qu'une seule action à la fois
- **Communication par échange de messages**
  - Aucune mémoire partagée
  - Les entrées des processus sont les messages reçus, les sorties sont les messages émis
- **Système asynchrone (souvent considéré):**
  - Asynchronisme des communications
    - Aucune hypothèse sur les temps d'acheminement des messages (Pas de borne supérieur)
  - Asynchronisme des traitements
    - Aucune hypothèse temporelle sur l'évolution des processus
- **Pas d'horloge commune**

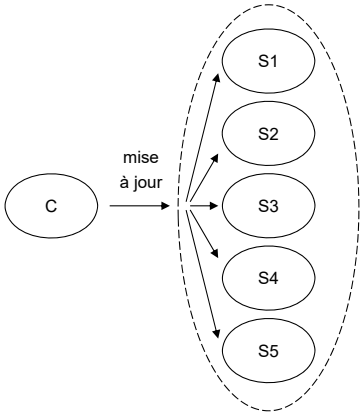
Protocoles de Diffusion

Motivation (1)

Dans certaines situations, les processus d'un système réparti (ou un sous-ensemble de ces processus) doivent être **adressés** comme **un tout**.

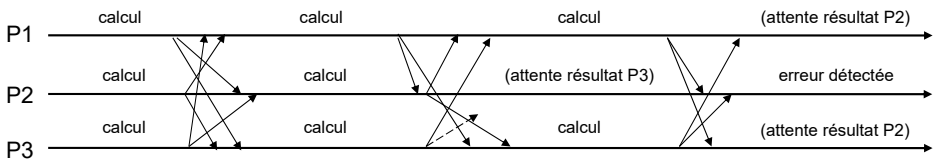


Application de calcul parallèle

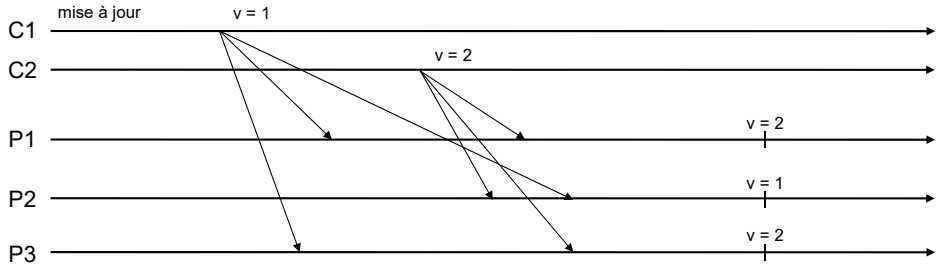


Serveur répliqué

Motivation (2)



Application de calcul parallèle : si un message est perdu, le calcul s'arrête.



Serveur répliqué : si les messages ne sont pas reçus dans le même ordre sur tous les processus, ils sont incohérents.

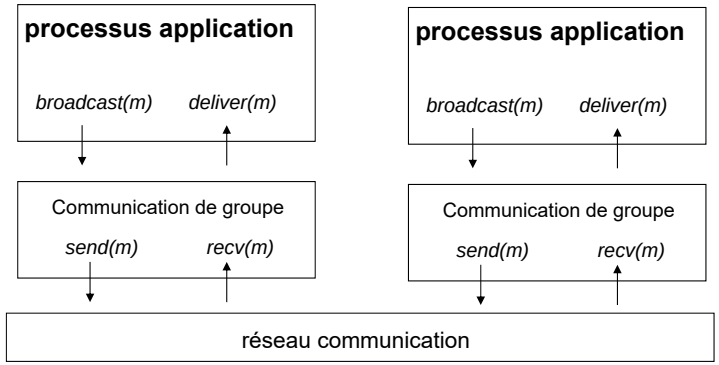
Diffusion : Définition

- **Un processus émetteur envoie un message à un *groupe* de processus.**
  - **Groupe** : ensemble de processus (les membres du groupe) auxquels on s'adresse par des diffusions, et non par des envois point à point.

Diffusion : primitives

- **Primitives de diffusion utilisées par le processus *p* :**
  - ***broadcast (m)*** : le processus ***p*** diffuse le message *m* au groupe.
  - ***deliver (m)*** : le message ***m*** est délivré au processus *p*.
- **La diffusion est réalisée au dessus d'un système de communication existant.**

Architecture

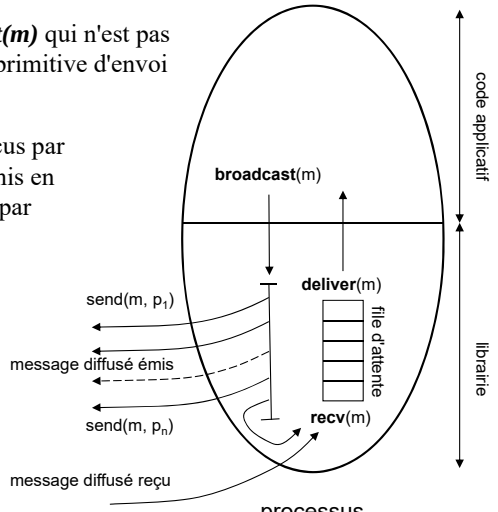


Modèle d'implémentation

• Les messages sont diffusés par ***broadcast(m)*** qui n'est pas atomique en pratique : elle s'appuie sur la primitive d'envoi point à point ***send(m, p).***

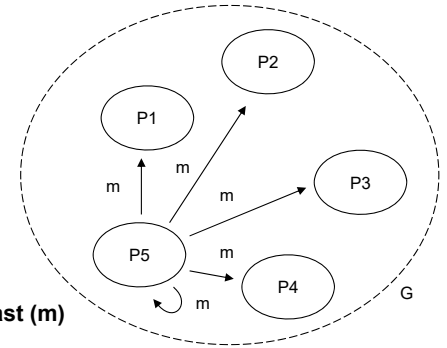
• Les messages ne sont pas directement reçus par l'application : ils sont reçus par ***recv(m)***, mis en attente, traités puis délivrés à l'application par ***deliver(m).***

• **Objectif** : implémenter des primitives ***broadcastX()*** et ***deliverX()*** qui garantissent la propriété X.



Diffusion: primitives (2)

- Le message envoyé à chaque processus est le même, mais le nombre et l'identité des destinataires est masqué à l'émetteur, qui les désigne par leur groupe d'appartenance. On assure ainsi la transparence de réplication.

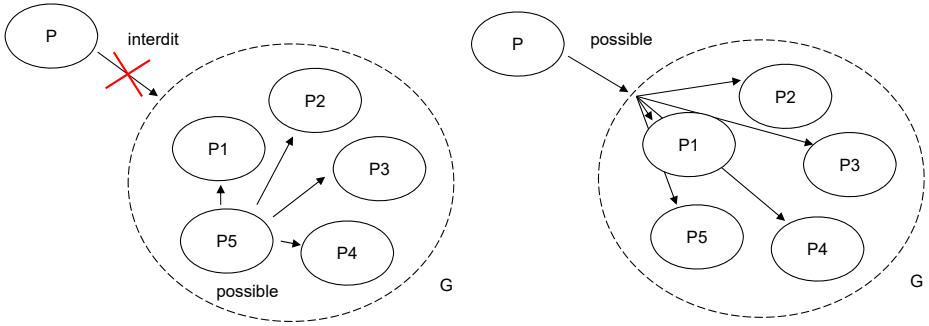


P5 exécute **broadcast (m)**

Groupe (1)

Un ***groupe*** peut être :

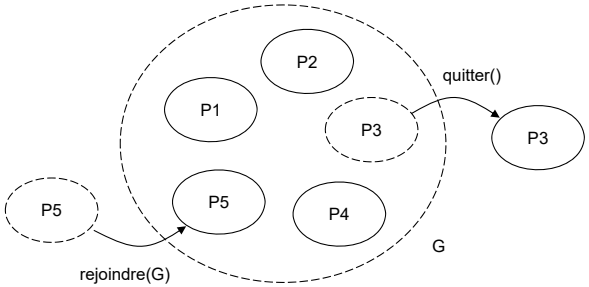
- **fermé** : ***broadcast(m)*** ne peut être appelé que par un membre du groupe
- **ouvert** : ***broadcast(m)*** peut être appelé par un processus extérieur au groupe



Groupe (2)

Un groupe peut être :

- **statique** : la liste des membres du groupe est fixe et déterminée au départ
- **dynamique** : les processus peuvent rejoindre ou quitter le groupe volontairement par l'intermédiaire d'un service de gestion de groupe



Groupe dynamique

Problèmes

- Les processus peuvent tomber en panne, notamment au milieu d'un envoi multiple de message
- L'ordre de réception des messages sur les différents processus destinataires n'est pas garanti (entrelancement dû aux latences réseau variables)

Problèmes à résoudre

- assurer des propriétés de diffusion :
  - garantie de **remise** des messages
  - garantie d'**ordonnancement** des messages reçus

Communications et Processus

- **Communications**
  - Point à point
  - Tout processus peut communiquer avec tout les autres
  - Canaux fiables **:** si un processus *p* correct envoie un message *m* à processus correct *q*, alors *q* finit par le recevoir ("eventually receives").
- **Processus**
  - Susceptibles de subir de pannes franches. Suite à une panne franche, un processus s'arrête définitivement : on ne considère pas qu'il puisse éventuellement redémarrer.

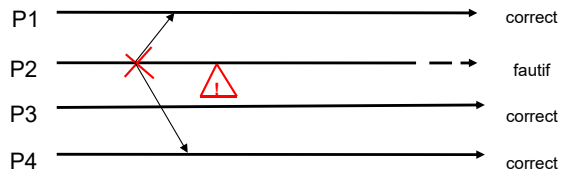
➡ Un processus qui ne tombe pas en panne sur toute une exécution donnée est dit **correct**, sinon il est dit **fautif**.

Propriétés des diffusions (1)

- **Garantie de remise**
  - Diffusion Best-effort (Best-effort Broadcast)
  - Diffusion Fiable (Reliable Broadcast)
  - Diffusion Fiable Uniforme (Uniform Reliable Broadcast).
- **Garantie d'ordonnancement**
  - les messages sont délivrés dans un ordre :
    - FIFO
    - Causal
    - Total
- Les garanties de remise et d'ordre sont orthogonales

1. Garantie de Remise: Best Effort

- **Diffusion Best-effort**
  - Garantie la délivrance d'un message à tous les processus corrects si l'émetteur est correct.
  - **Problème :** pas de garantie de remise si l'émetteur tombe en panne



P2 tombe en panne avant d'envoyer le messages à P3

Diffusion Best-Effort

- **Spécification**
  - **Validité** : si p_1 et p_2 sont corrects alors un message *m* diffuser par p_1 finit par être délivré par p_2.
  - **Intégrité**: un message *m* est délivré au plus une fois et seulement s'il a été diffusé par un processus.
- **Algorithme**

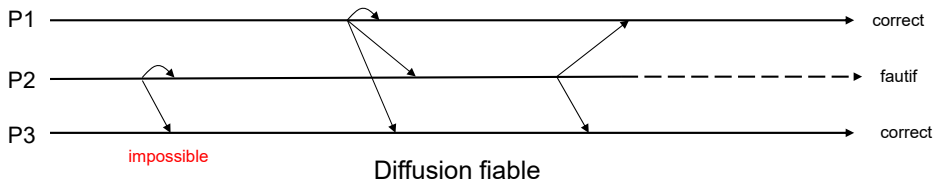
Processus ***P :***

BestEffort_broadcast (m)
. envoyer m à tous les processus y compris p /* groupe fermé */

upon recv(m) :
BestEffort_deliver(m) /* délivrer le message */

2. Garantie de Remise : Fiable

- **Diffusion Fiable (Reliable Broadcast)**
  - si l'émetteur du message *m* est **correct**, alors **tous** les destinataires **corrects** délivrent le message ***m.***
  - si l'émetteur du message *m* est ***fautif***, tous ou aucun processus corrects délivrent le message ***m.***



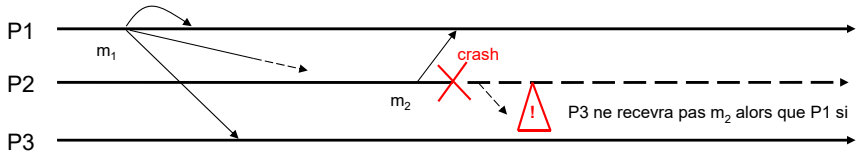
Diffusion fiable

Diffusion Fiable – spécification (1)

- **Spécification**
  - **Validité** : si un processus **correct** diffuse le message *m*, alors tous les processus **corrects** délivrent *m*
  - **Accord** : si un processus **correct** délivre le message *m*, alors tous les membres **corrects** délivrent *m*
  - **Intégrité**: Un message *m* est délivré au plus une fois à tout processus **correct**, et seulement s'il a été diffusé par un processus.

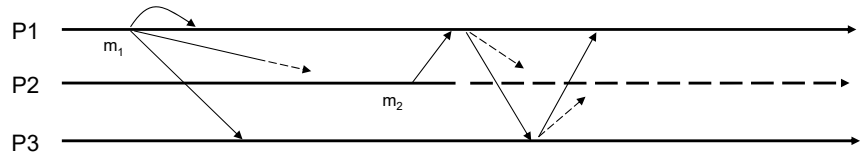
Diffusion Fiable : principe

Si un processus correct délivre le message diffusé m, alors tout processus correct délivre aussi *m*



Diffusion Fiable : principe

Implémentation *possible* : sur réception d'un message diffusé par un autre processus, chaque processus rediffuse ce message avant de le délivrer.



Diffusion Fiable : algorithme

Chaque message m est estampillé de façon unique avec :
- *sender (m)* : l'identité de l'émetteur
- *seq#(m)* : numéro de séquence

Processus ***P :***

Variable locale :
rec = ø;

Real_broadcast (m)
estampiller *m* avec *sender(m)* et *seq#(m);*
envoyer *m* à tous les processus y compris *p*

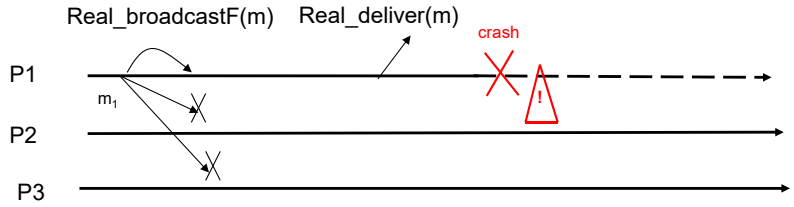
upon recv(m) do
if m ∉ rec then
rec ∪= { m }
if sender(m) != p **then**
envoyer *m* à tous les processus sauf p
Real_deliver(m) /* délivrer le message */

Diffusion Fiable : discussion

- Avantages :
  - la fiabilité ne repose pas sur la détection de la panne de l'émetteur
  - l'algorithme est donc valable dans tout modèle temporel
- Inconvénients :
  - l'algorithme est très inefficace : il génère n (n – 1) envois par diffusion
  - ce qui le rend inutilisable en pratique
- Remarques :
  - l'algorithme ne garantit aucun ordre de remise

Diffusion Fiable

- Problème **:**
  - aucune garantie de délivrance est offerte pour les processus fautifs
    - Exemple :



- P_1 délivre *m* et après il crash ; P_2 et P_3 ne reçoivent pas *m*
- P_1 avant sa défaillance peut exécuter des actions irréversibles comme conséquence de la délivrance de *m*

3: Garantie de Remise – fiable uniforme

- **Diffusion Fiable Uniforme (Uniform Reliable Broadcast)**
  - Si un message *m* est délivré par un processus (**fautif** ou **correct**), alors tout processus **correct** finit aussi par délivré *m*.

Diffusion Fiable Uniforme

- **Propriété d'uniformité**
  - Une propriété (accord, intégrité) est dite **uniforme** si elle s'applique à tous les processus : **corrects** et **fautifs**.
- **Diffusion Fiable Uniforme**
  - **Validité :** si un processus correct diffuse le message *m*, alors tous les processus corrects délivrent *m*
  - **Accord uniforme** : si un processus (**correct** ou **fautif**) délivre le message *m*, alors tous les membres corrects délivrent *m*.
  - **Intégrité uniforme**: Un message *m* est délivré au plus une fois à tout processus (**correct** ou **fautif**), et seulement s'il a été diffusé par un processus.

Diffusion Fiable Temporisée

- **Diffusion fiable temporisée = diffusion fiable + borne**
  - Système de communication synchrone
  - **Borne** : il existe une constante Δ telle que si un message *m* est diffusé à l'instant *t*, alors aucun processus correct ne délivre *m* après le temps $t+\Delta$.

Garantie d'ordre (1)

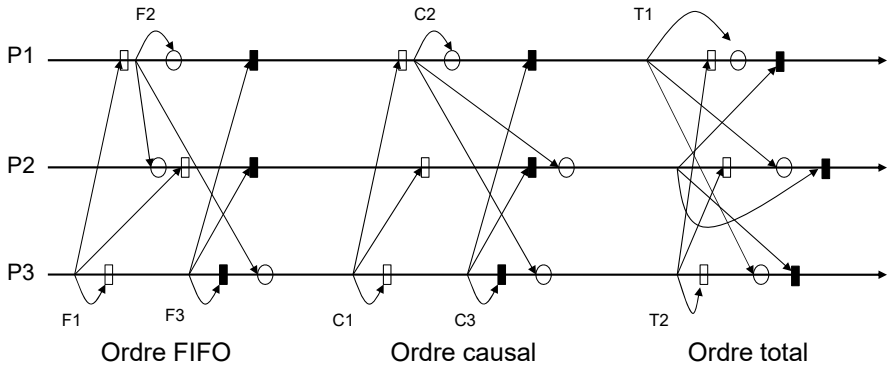
- Ordre Total
  - Les messages sont délivrés dans le même ordre à tous leurs destinataires.
- Ordre FIFO
  - si un membre diffuse m_1 puis m_2, alors tout membre correct qui délivre m_2 délivre m_1 avant m_2.
- Ordre Causal
  - si *broadcast*(m_1) précède causalement *broadcast* (m_2), alors tout processus correct qui délivre m_2, délivre m_1 avant m_2.

Observations :

- La propriété d'ordre total est indépendante de l'ordre d'émission
- Les propriétés d'ordre FIFO et Causal sont liées à l'ordre d'émission

Garantie d'ordre (2)

- **Exemple**

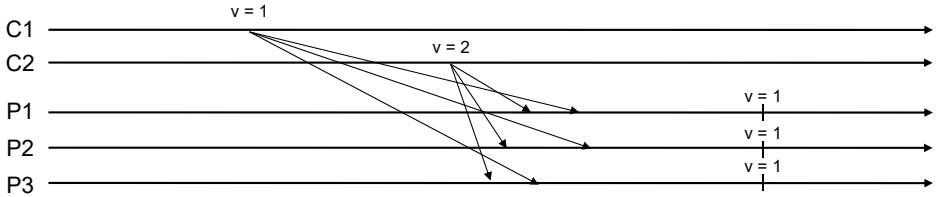


Garantie d'ordre (3)

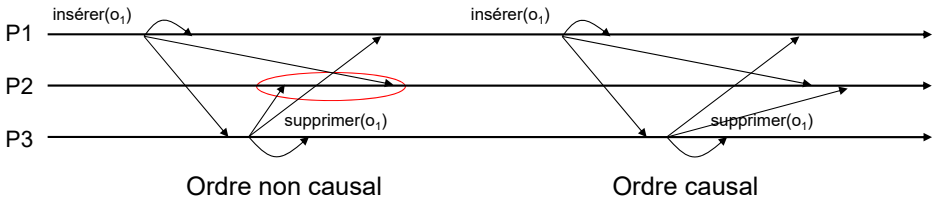
- Remarques :
  - une diffusion causale est nécessairement FIFO (la diffusion causale peut être vue comme une généralisation de l'ordre FIFO à tous les processus du groupe)
  - L'ordre FIFO et l'ordre causal ne sont que des ordres partiels : ils n'imposent aucune contrainte sur l'ordre de délivrance des messages diffusés concurremment
  - l'ordre total n'a pas de lien avec l'ordre FIFO et l'ordre causal : il est à la fois plus fort (ordre total des messages délivrés) et plus faible (aucun lien entre l'ordre de diffusion et l'ordre de délivrance)

Garantie d'ordre - Exemple utilisation (4)

Ordre total : permet de maintenir la cohérence des répliques d'un serveur en présence d'écrivains multiples.

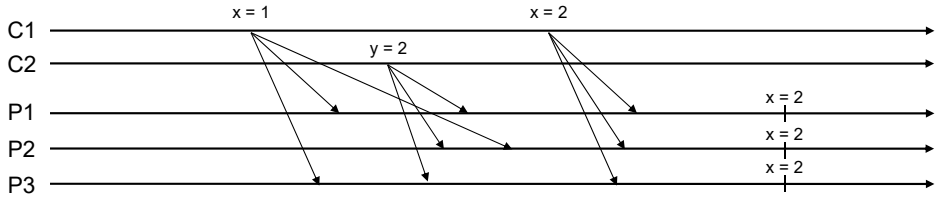


Ordre causal : permet de préserver à faible coût l'enchaînement d'opérations logiquement liées entre elles.



Garantie d'ordre - Exemple utilisation (5)

Ordre FIFO : permet de maintenir la cohérence des répliques d'un serveur en présence d'un écrivain unique.

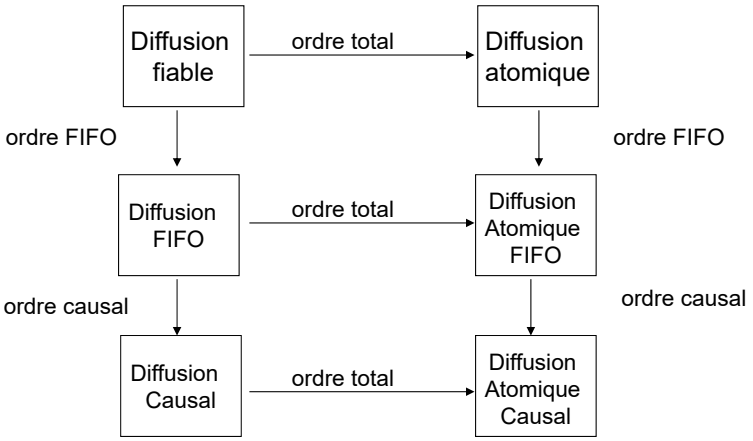


Les trois garanties d'ordre FIFO, causal et total sont plus ou moins coûteuses à implémenter : choisir celle juste nécessaire à l'application visée.

Types de Diffusion Fiable (1)

- Diffusion FIFO = Diffusion fiable + Ordre FIFO
- Diffusion Causal (CBCAST) = Diffusion fiable + Ordre Causal
- Diffusion Atomique (ABCAST) = Diffusion fiable + Ordre Total
- Diffusion Atomique FIFO = Diffusion FIFO + Ordre Total
- Diffusion Atomique Causal = Diffusion Causal + Ordre Total

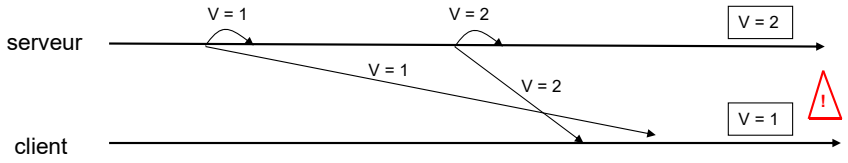
Types de Diffusion Fiable (2)



Relation entre les primitives de diffusion [Hadzilacos & Toueg]

Diffusion FIFO - motivation

- **Dans la diffusion fiable il n'y a aucune spécification sur l'ordre de délivrance des messages.**



Diffusion FIFO

- **Diffusion FIFO = diffusion fiable + ordre FIFO**
  - **Ordre FIFO** : si un membre diffuse m_1 puis m_2, alors tout membre correct qui délivre m_2 délivre m_1 avant m_2.
  - Ayant un algorithme de diffusion fiable, il est possible de le transformer dans un algorithme de diffusion FIFO

Diffusion FIFO – algorithme (1)

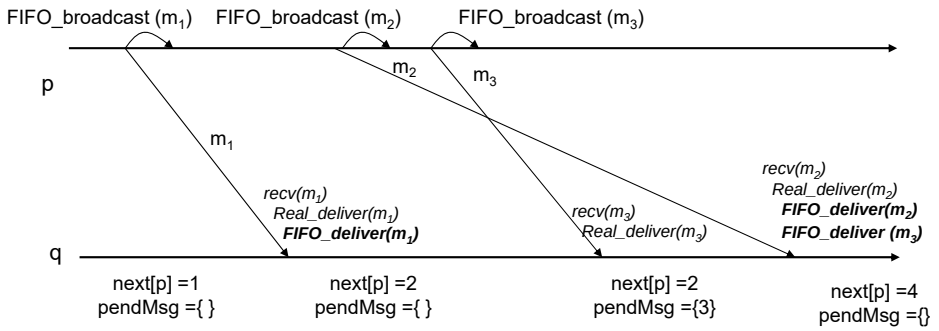
Processus ***p :***

Variable locale :
pendMsg = ø; /* message pas encore délivré */
next [N] = 1 pour tous processus; /*seq# du prochain message de *q* que *p* doit délivrer */

FIFO_broadcast (m)
Real_broadcast(m); /*m estampillé avec seq# */

upon Real_deliver(m) do
s = sender (m);
pendMsg ∪= { m }
while (∃ m' ϵ PendMsg : sender(m') =s and seq#(m') = next[s]) **do**
FIFO_delivrer(m') /* délivrer le message */
next[s]++;
pendMsg -= { m' };

Diffusion FIFO – algorithme (2)

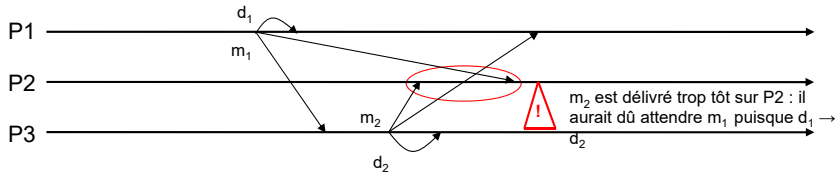


Diffusion Causal - CBCAST

- **Diffusion Causal = diffusion fiable + ordre Causal**
  - Objectif : délivrer les messages dans l'ordre causal de leur diffusion.
  - **Ordre causal** : si *broadcast*(m_1) précède causalement *broadcast* (m_2), alors tout processus correct qui délivre m_2, délivre m_1 avant m_2.
    - **$broadcast_p(m_1) \rightarrow broadcast_q(m_2) \Leftrightarrow deliver_p(m_1) \rightarrow deliver_q(m_2)$**
  - Causal Order ⟶ FIFO order
  - Fifo Order $\not\rightarrow$ Causal Order

Diffusion Causal

$broadcast_p(m_1) \rightarrow broadcast_q(m_2) \Leftrightarrow deliver_p(m_1) \rightarrow deliver_q(m_2)$



➢ Un algorithme de diffusion FIFO peut être transformé dans un algorithme de diffusion causal :

• transporter avec chaque message diffusé l'historique des messages qui le précèdent causalement.

Diffusion Causal – algorithme

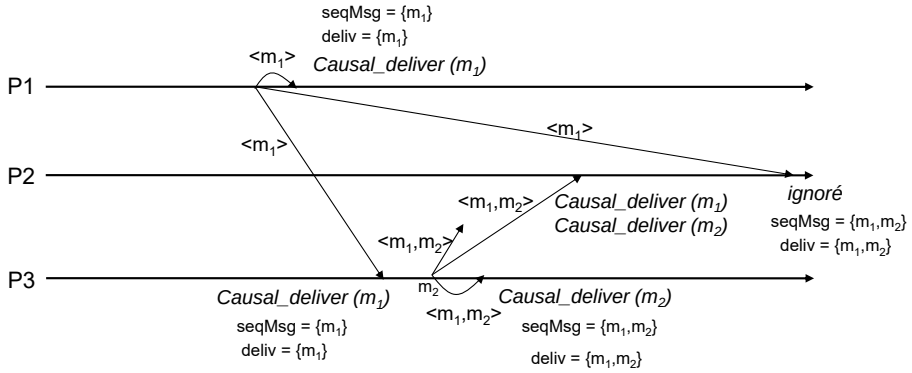
Processus ***p :***

Variable locale :
seqMsg = vide; /* sequence de messages que *p* a délivré depuis sa diffusion précédente */
delv = ø; /* messages délivres */

Causal_broadcast (m)
FIFO_broadcast(seqMsg Θ m); /* diffuser tous les messages délivrés depuis la diffusion précédente + m */
seqMsg = vide;

upon FIFO_deliver(m_1,m_2,.. m_n) do
for i=1.. n **do**
if $m_i \notin$ delv **then**
Causal_delivrer(m_i) /* délivrer le message */
delv ∪= { m_i }
seqMsg Θ= m_i /*ajouter m_i à la fin de la seqMsg */

Diffusion Causal – algorithme



- Avantage :
  La délivrance d'un message n'est pas ajournée en attente d'une condition
- Inconvénient
  - Taille des messages

Diffusion Causal – algorithme avec horloges vectorielles (sans garantie de remise)

➢ Historique de messages peut être représenté au moyen d'une d'horloge vectorielle

Processus ***P :***

$HV[k]_m$ venant de P_j représente :
- k = j : le nombre de messages diffusés par P_j
- k ≠ j : le nombre de diffusions de P_k délivrées par P_j avant diffusion de m.

Variables locales :
HV[N] = {0, 0, …, 0}
FA = ∅

Causal_Broadcast(m)
HV[i] += 1
estampiller m avec HV;
envoyer m à tous les processus y compris p

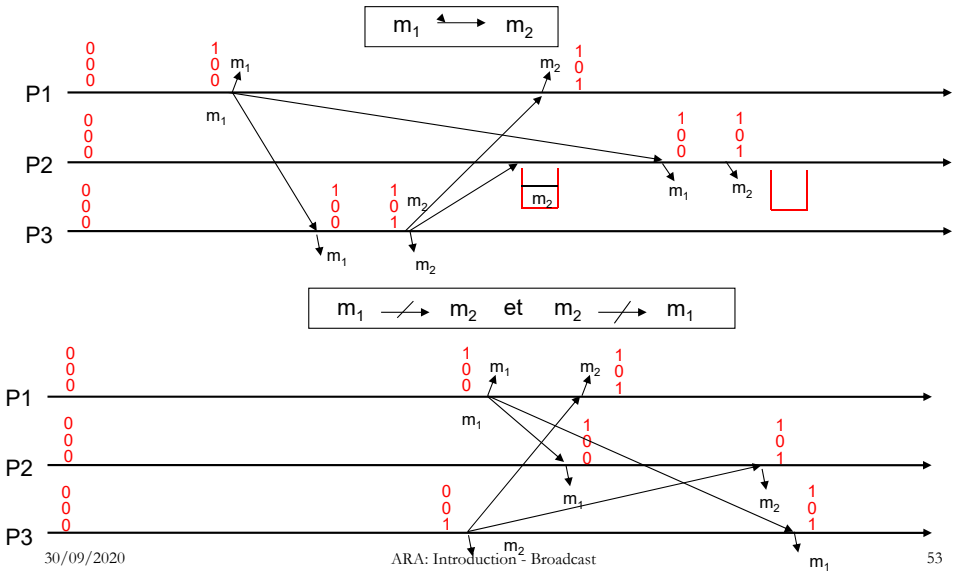
Isis - Birman 91

Diffusion Causal – algorithme avec horloges vectorielles (sans garantie de remise)

Upon recv(m, $HV[]_m$) :
s = sender (m);
FA.queue(< m, $HV[]_m$ >)
delay delivery of m **until**
(1) $HV[s]_m = HV[s]_p + 1$ **and** (2) $HV[k]_m \leq HV[k]_p$ pour tout k; k ≠ s
// D'autres réceptions se produisent pendant l'attente. On attend d'avoir délivré :
// 1- toutes les diffusions précédentes effectuées par *s*,
// 2- toutes les diffusions délivrées par *s* avant la diffusion de m
FA.dequeue(< m, $HV[]_m$ >)
deliver(m);
$HV[s]_p$ +=1;

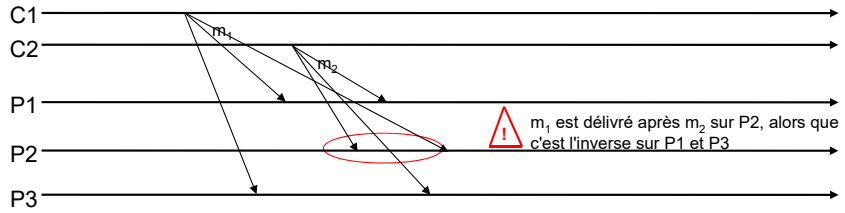
- (1) : assure que *p* a délivré tous les messages provenant de *s* qui précédent *m*
- (2) : assure que *p* a délivré tous les messages délivrés par *s* avant que celui-ci envoie *m*

Diffusion Causal – algorithme avec horloges vectorielles - Exemple



Diffusion Atomique - ABCAST

- **Diffusion atomique = diffusion fiable + ordre total**
  - Tous les processus corrects délivrent le même ensemble de messages dans le même ordre.
  - **Ordre Total** : si les processus corrects *p* et *q* délivrent tous les deux les messages *m* et *m*', alors *p* délivre *m* avant *m'* seulement si *q* délivre *m* avant *m*'.
  - Exemple d'une diffusion **pas** atomique



Diffusion Atomique - ABCAST

➢ **Résultat fondamental** : Dans un système asynchrone avec pannes franches, la diffusion atomique est équivalent au consensus.

Consensus impossible dans un système asynchrone avec pannes franches ⇨ Diffusion atomique impossible dans un système asynchrone avec pannes franches

❑. Si on dispose d'un algorithme de diffusion atomique, on sait réaliser le consensus
- Chaque processus diffuse atomiquement sa valeur proposée à tous les processus
- Tous les processus reçoivent le même ensemble de valeurs dans les même ordre
- Ils décident la première valeur

❑ Si on dispose d'un algorithme de consensus, on sait réaliser la diffusion atomique

Diffusion Atomique ⟺ Consensus

Chandra & Toueg 1996

Diffusion Atomique - ABCAST

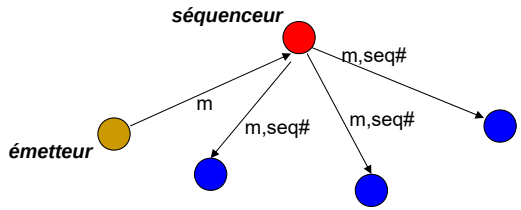
- **Remarques :**
  - ABCAST n'est pas réalisable dans un système asynchrone si on suppose l'existence de fautes (d'après FLP).
  - ABCAST est réalisable (n nodes):
    - Avec un détecteur de pannes de classe P ou S en tolérant n-1 pannes
    - Avec détecteur de pannes de classe ◊S en tolérant n/2 -1 pannes
    - Avec un protocole de diffusion fiable temporisée en utilisant des hypothèse de synchronisme.

Diffusion Atomique - algorithmes

- **Un protocole ABCAST doit garantir l'ordre de remise de messages et tolérer les défaillances**
- **L'ordre d'un protocole ABCAST peut être assuré par :**
  - Un ou plusieurs séquenceurs
    - séquenceur fixe
    - séquenceur mobile
  - Les émetteurs
    - À base de privilège
  - Les récepteurs
    - Accord des récepteurs

- Remarques: les algorithmes présentés à la suite ne traitent pas les pannes

Diffusion totalement ordonnée : Séquenceur fixe



Diffusion totalement ordonnée : Séquenceur fixe

- **Principe :**
  - Un processus, le séquenceur, est choisi parmi tous les processus
    - Responsable de l'ordonnancement des messages
  - Émetteur envoie le message *m* au séquenceur
    - Séquenceur attribue un numéro de séquence *seq#* à *m*
    - Séquenceur envoie le message à tous les processus.

Séquenceur fixe - algorithme

Processus ***P :***

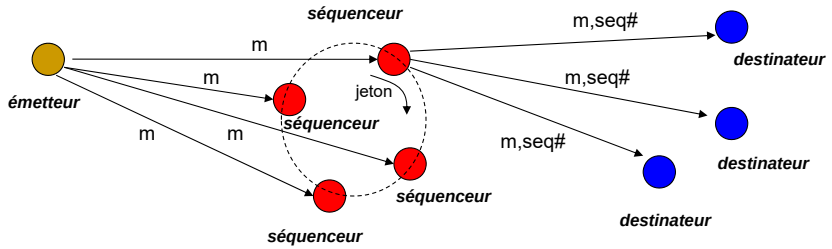
Variables locales :
nextdelv = 1;
pend = ø;

Emetteur :
OT_broadcast (m)
send m au séquenceur;

Séquenceur :
intit :
seq#=1;
upon revc(m) do
send (m,seq#) à to processus
seq#++;

Destinateur :
upon revc(m) do
pend ∪= {m}
while (∃ (m',seq#') ϵ pend : seq#'=nextdelv**) do**
OT_deliver (m')
nextdelv++;
pend -= {m'}

Diffusion totalement ordonnée : Séquenceur mobile



Diffusion totalement ordonnée : Séquenceur mobile

- **Principe**
  - Un groupe de processus agissent successivement comme séquenceur
  - Un message est envoyé à tous les séquenceurs.
  - Un jeton circule entre les séquenceurs, contenant :
    - un numéro de séquence
    - Liste de messages déjà séquencés
  - Lors de la réception du jeton, un séquenceur :
    - attribue un numéro de séquence à tous les messages pas encore séquencés et envoie ces messages aux destinateurs
    - Ajoute les messages envoyés dans la liste du jeton
- **Avantages**
  - répartition de charge
- **Inconvénients**
  - Taille jeton
  - coût circulation du jeton

Séquenceur mobile - algorithme

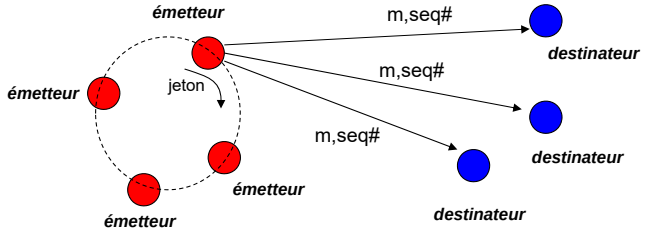
Variables locales :
nextdelv = 1;
pend = ø;

Emetteur :
OT_broadcast (m)
send m à tous les séquenceurs;

Destinateur :
upon revc(m) do
pend ∪= {m}
while (∃ (m',seq#') ϵ pend : seq#'=nextdelv**) do**
OT_deliver (m')
nextdelv++;
pend -={m'}

Séquenceur :
init :
rec = ø;
if (p= s_1)
token.seq# = 1
token.liste = ø;
upon revc(m) do
rec ∪= {m}
upon recv(token) do
for each m' in rec \ token.liste do
send (m',token.seq#) à tous les destinateurs
token.seq#++;
token.liste ∪= {m}
send (token) au prochain séquenceur

Diffusion totalement ordonnée : à base de priorité



Diffusion totalement ordonnée : à base de priorité

- **Principe**
  - Un jeton donne le droit d'émettre
  - Jeton circule entre les émetteurs contenant le numéro de séquence du prochain message à envoyer.
  - Lorsqu'un émetteur veut diffuser un message, il doit attendre avoir le jeton
    - attribue un numéro de séquence aux messages à diffuser
    - envoie le jeton aux prochains émetteurs
- **Inconvénients**
  - Nécessaire de connaître les émetteurs (pas adéquat pour de groupe ouvert)
  - Pas très équitable : un processus peut garde le jeton et diffuser un nombre important de messages en empêchant les autres de le faire

Diffusion totalement ordonnée : à base de priorité

Variables locales :
nextdelv = 1;
pend = ø;
send_pend = ø;

Destinateur :
upon revc(m) do
pend ∪= { m}
while (∃ (m',seq#') ϵ pend : seq#'=nextdelv**) do**
OT_deliver (m')
nextdelv++;
pend -={m'}

Emetteur :
init :
send_pend = ø;
if (p=s1)
token.seq# = 1

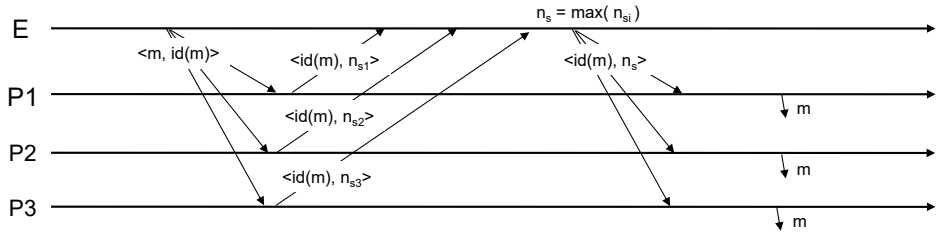
procedure OT_broadcast (m)
send_pend ∪= {m}

upon recv(token) do
for each m' in send_pend do
send (m',token.seq#) à tous les destinateurs
token.seq#++;
send_pend = ø;
send (token) au prochain émetteur

Diffusion totalement ordonnée : accord récepteurs

- **Principe**
  - Les processus se concertent pour attribuer un numéro de séquence à chaque message. Chaque diffusion nécessite deux phases :
    - diffusion du message et collecte des propositions de numérotation
    - choix d'un numéro définitif et diffusion du numéro choisi

Accord récepteurs



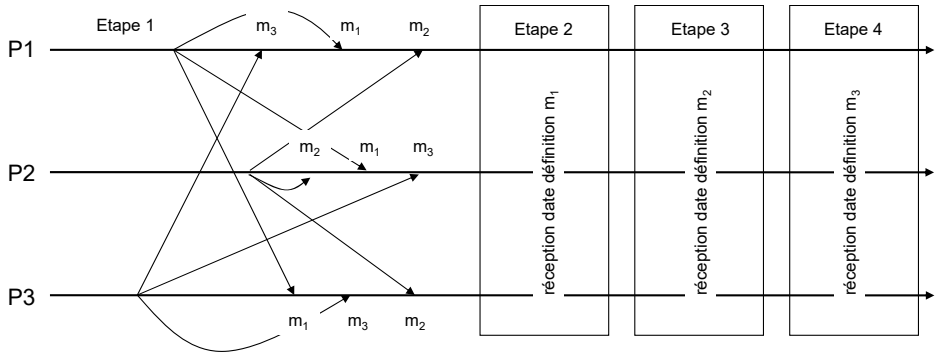
Les numéros proposés sont <*date logique réception, identité récepteur*> pour assurer un ordre total. Chaque processus maintient une file d'attente des messages en attente de numérotation définitive, triée de façon croissante sur les numéros.

Accord récepteurs : algorithme

- *E* diffuse le message *m* au groupe :
  - sur réception de *m*, P_j attribue à *m* son numéro de réception provisoire, le marque ***non délivrable***, et l'insère dans sa file d'attente
  - puis P_j renvoie à *E* le numéro provisoire de *m* comme proposition de numéro définitif
  - quand *E* a reçu tous les numéros proposés, il choisit le plus grand comme numéro définitif et le rediffuse
  - sur réception du numéro définitif, P_j réordonne *m* dans sa file et le marque délivrable
  - puis P_j délivre tous les messages marqués ***délivrable*** situés en tête de la file d'attente

Birman - Joseph 87

Accord récepteurs : exemple



P1, P2 et P3 diffusent simultanément les trois messages m_1, m_2 et m_3 (seuls les messages de l'étapes 1 sont représentés).

Note : il s'agit d'un **exemple** d'exécution ; la date définitive d'un message n'arrive **pas nécessairement** dans le même laps de temps sur tous les processus, ni dans le même ordre pour les différents messages.

Accord récepteurs : exemple (cont.)

Etape 1 : réception des messages et proposition de numérotation

FA P1			FA P2			FA P3		
m_3 15.1 N	m_1 16.1 N	m_2 17.1 N	m_2 16.2 N	m_1 17.2 N	m_3 18.2 N	m_1 17.3 N	m_3 18.3 N	m_2 19.3 N

Etape 2 : réception de la date de définitive de m_1 : 17.3

FA P1			FA P2			FA P3		
m_3 15.1 N	m_2 17.1 N	m_1 17.3 D	m_2 16.2 N	m_1 17.3 D	m_3 18.2 N	m_1 17.3 D	m_3 18.3 N	m_2 19.3 N

m_1 est délivré sur P3

Accord récepteurs : exemple (cont.)

Etape 3 : réception de la date de définitive de m_2 : 19.3

FA P1			FA P2			FA P3	
m_3 15.1 N	m_1 17.3 D	m_2 19.3 D	m_1 17.3 D	m_3 18.2 N	m_2 19.3 D	m_3 18.3 N	m_2 19.3 D

m_1 est délivré sur P2

Etape 4 : réception de la date de définitive de m_3 : 18.3

FA P1			FA P2		FA P3	
m_1 17.3 D	m_3 18.3 D	m_2 19.3 D	m_3 18.3 D	m_2 19.3 D	m_3 18.3 D	m_2 19.3 D
m_1, m_3 puis m_2 délivrés sur P1			m_3 puis m_2 délivrés sur P2		m_3 puis m_2 délivrés sur P3	

Diffusion totalement ordonnée tolérance aux fautes

- **Quelques mécanismes :**
  - Détecteurs de défaillance
  - Redondance
    - Exemple : séquenceur
  - Stabilité des messages
    - Un message est *k-stable* s'il a été reçu par k processus.
      - *f* défaillances : un messages *(f+1)-stable* a été reçu par au moins 1 processus correct. Sa délivrance peut être garantie.
  - Pertes de messages
    - Numérotation des messages.

Bibliographie

- X. Défago and A. Schiper and P. Urban Total order broadcast and multicast algorithms: Taxonomy and survey, *ACM Comput. Surv., 36(4):372—421*.
- K.Birman, T. Joseph. Reliable communication in presence of failures. *ACM Transactions on Computer Systems,* Vol. 5, No. 1, Feb. 1987
- K. Birman and R. Cooper. The ISIS Project: Real Experience with a Fault Tolerant Programming System. *Operating Systems Review,* Apr. 1991, pages 103-107.
- K. Birman, A. Schiper and P. Stephenson. Lightweight Causal and Atomic Group Multicast. *ACM Transactions on Computer Systems*, Aug. 1991, (3):272-314.
- R. Guerraoui, L. Rodrigues. *Reliable Distributed Programming*, Springer, 2006
- V. Hadzilacos and S. Toueg. A Modular Approach to Fault-tolerant Broadcasts and Related Problems. *Technical Report TR94-1425*. Cornell University.
- T.Chandra and S.Toueg. Unreliable failure detectors for reliable distributed systems, *Journal of the ACM,Vol.43*. No.2, 1996, pages 225-267.