
ARA

Algorithmique Répartie avancée

Master 2 - SAR

Luciana Arantes

17/09/2016

ARA: Introduction - Broadcast

1

Planning

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

17/09/2016

ARA: Introduction - Broadcast

2

Planning

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

17/09/2016

ARA: Introduction - Broadcast

3

Rappels

Modèles de fautes et modèles temporels

17/09/2016

ARA: Introduction - Broadcast

4

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

17/09/2016

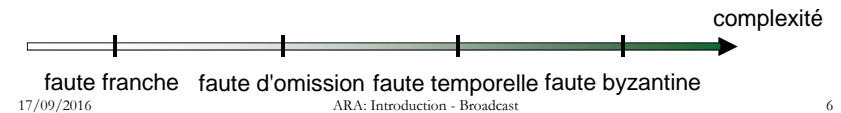
ARA: Introduction - Broadcast

5

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

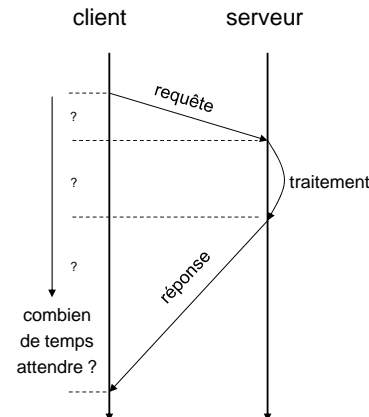
- vitesses processus différentes
- délais de transmission variables

■ 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



17/09/2016

ARA: Introduction - Broadcast

7

Modèles temporels (2)

Modèle temporel = hypothèses sur :

- délais de transmission des messages
- écart entre les vitesses des processus

système synchrone :

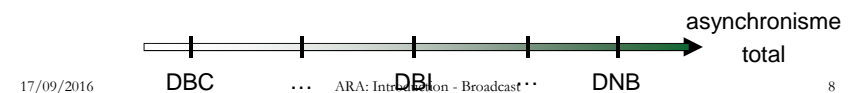
Modèle Délais/écarts Bornés Connus (DBC)
- permet la détection parfaite de faute

système partiellement synchrone :

Modèle Délais/écarts Bornés Inconnus (DBI)

système asynchrone :

Modèle Délais/écarts Non Bornés (DNB)

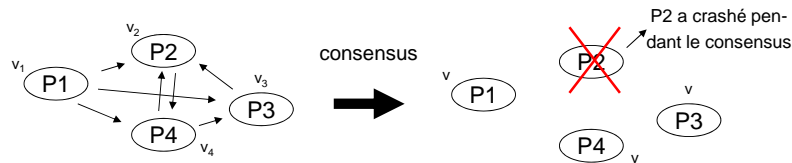


Modèles temporels (3)

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):

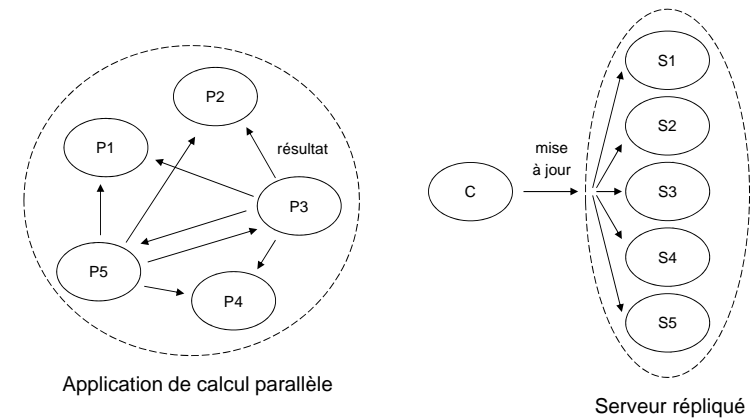
- Asynchronisme des communications
 - Aucune hypothèse sur les temps d'acheminement des messages (Pas de borne supérieur)
- Asynchronisme des traitements
 - Aucune hypothèse temporelle sur l'évolution des processus

■ Pas d'horloge commune

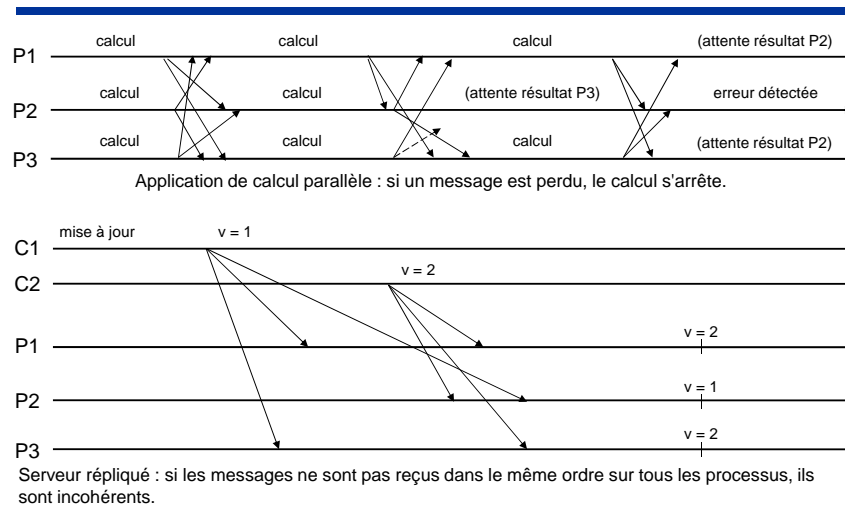
Protocoles de Diffusion

Motivation (1)

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



Motivation (2)



17/09/2016

ARA: Introduction - Broadcast

13

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.

17/09/2016

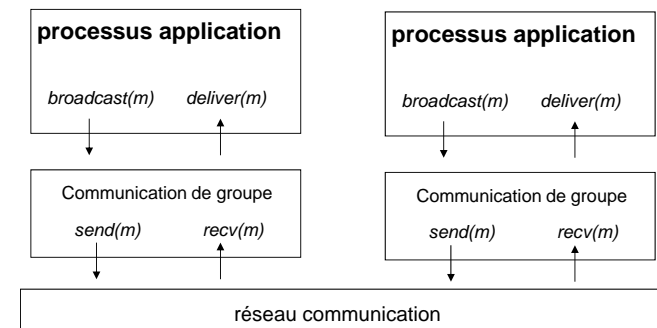
ARA: Introduction - Broadcast

14

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



17/09/2016

ARA: Introduction - Broadcast

15

17/09/2016

ARA: Introduction - Broadcast

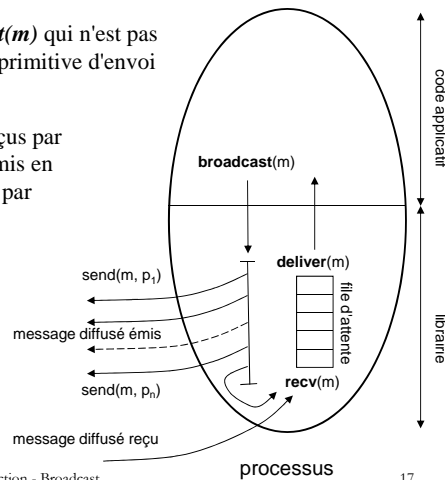
16

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.



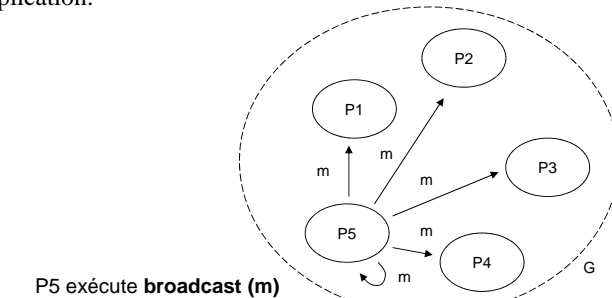
17/09/2016

ARA: Introduction - Broadcast

17

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.



17/09/2016

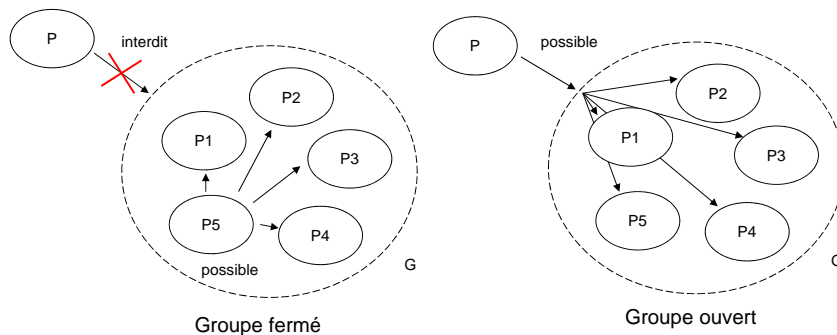
ARA: Introduction - Broadcast

18

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



17/09/2016

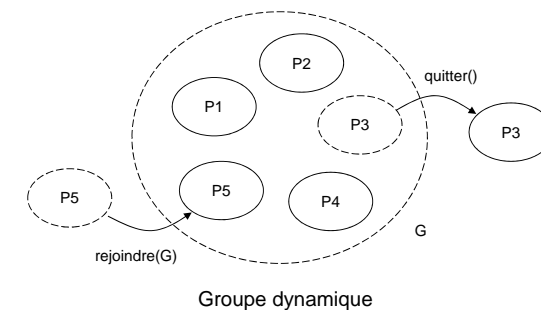
ARA: Introduction - Broadcast

19

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



17/09/2016

ARA: Introduction - Broadcast

20

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

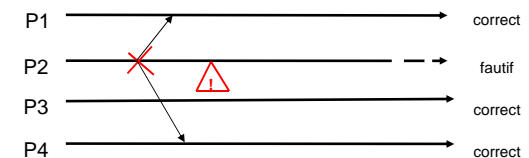
- les messages sont délivrés dans un ordre :
 - FIFO
 - Causal
 - Total

■ 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 */

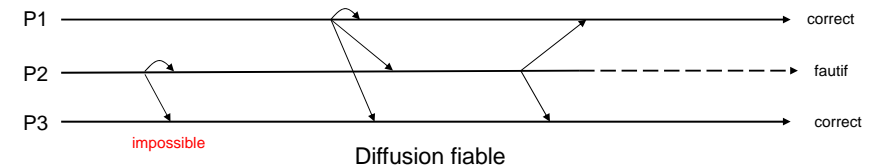
upon rcv(m) :

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 .



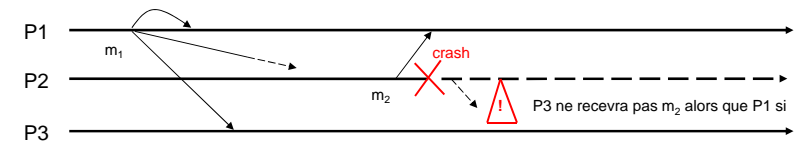
Diffusion Fiable – spécification (1)

■ 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.

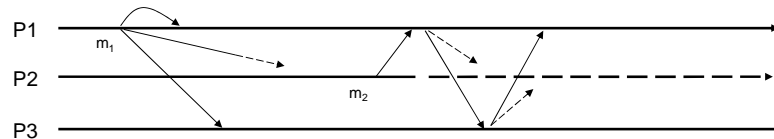
Diffusion Fiable : principe

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 :

$rec = \emptyset$;

Real_broadcast (m)

estampiller m avec $sender(m)$ et $seq\#(m)$;

envoyer m à tous les processus y compris p

upon rcv(m) do

if $m \notin rec$ then

$rec \cup = \{ m \}$

if $sender(m) \neq p$ 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 :

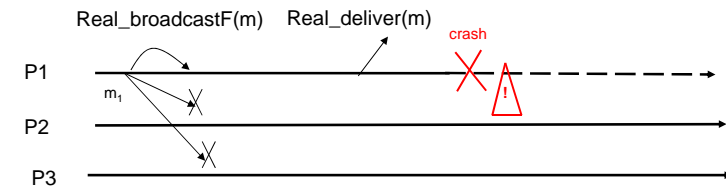
- l'algorithme ne garantit aucun ordre de remise

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 délivrer 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)

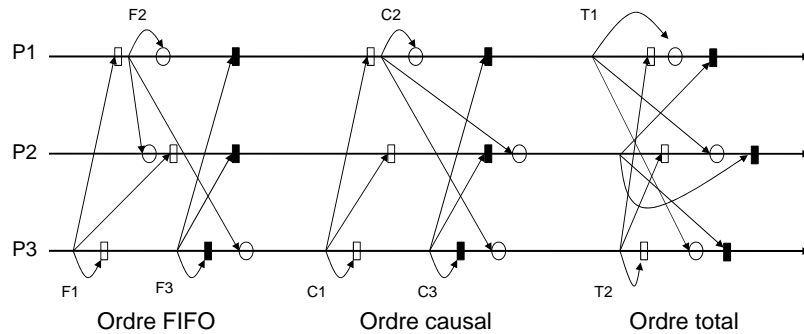
- **Ordre Total**
 - Les messages sont délivrés dans le même ordre à tous leurs destinataires.
- **Ordre FIFO**
 - si un membre diffuse m_1 puis m_2 , alors tout membre correct qui délivre m_2 délivre m_1 avant m_2 .
- **Ordre Causal**
 - si $\text{broadcast}(m_1)$ précède causalement $\text{broadcast}(m_2)$,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



17/09/2016

ARA: Introduction - Broadcast

37

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)

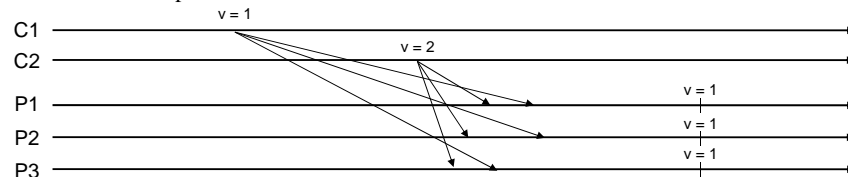
17/09/2016

ARA: Introduction - Broadcast

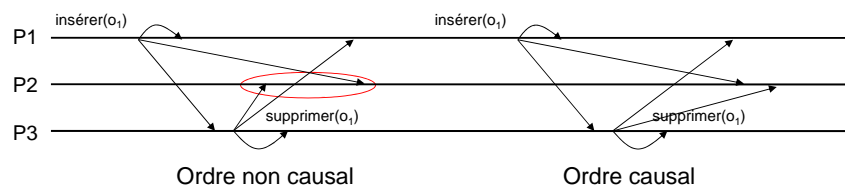
38

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.



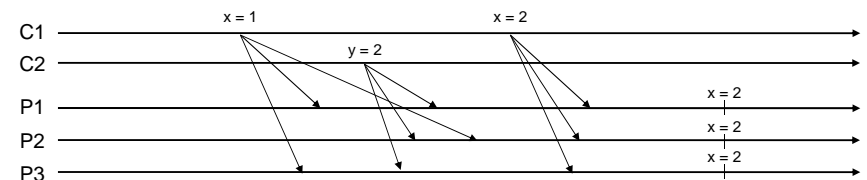
17/09/2016

ARA: Introduction - Broadcast

39

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.

17/09/2016

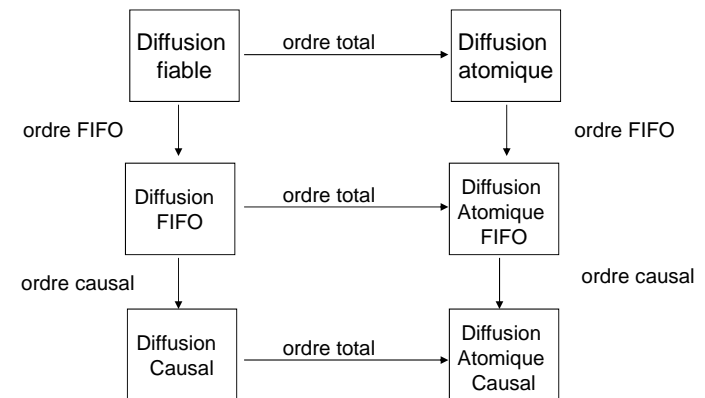
ARA: Introduction - Broadcast

40

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

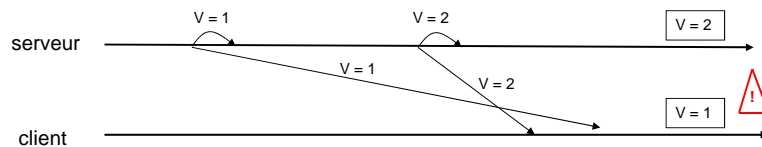
Types de Diffusion Fiable (2)



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 = \emptyset ; /* 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 \cup = { m }

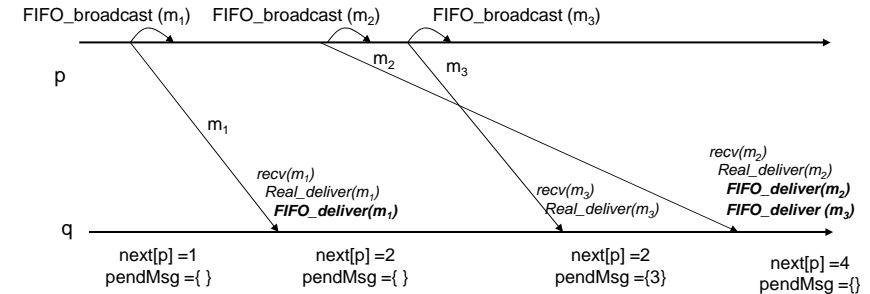
while ($\exists m' \in \text{PendMsg} : \text{sender}(m') = s \text{ and } \text{seq\#}(m') = \text{next}[s]$) **do**

FIFO_delivrer(m') /* délivrer le message */

next[s]++;

pendMsg -= { m' };

Diffusion FIFO – algorithme (2)



Diffusion Causal - CBCAST

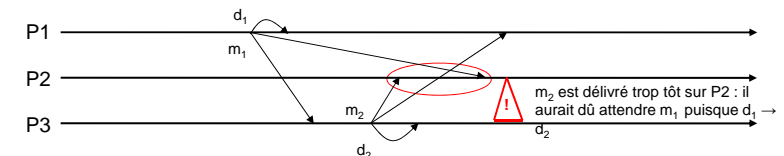
■ Diffusion Causal = diffusion fiable + ordre Causal

- Objectif : délivrer les messages dans l'ordre causal de leur diffusion.
- **Ordre causal** : si $\text{broadcast}(m_1)$ précède causalement $\text{broadcast}(m_2)$, alors tout processus correct qui délivre m_2 , délivre m_1 avant m_2 .
 - $\text{broadcast}_p(m_1) \rightarrow \text{broadcast}_q(m_2) \Leftrightarrow \text{deliver}_p(m_1) \rightarrow \text{deliver}_q(m_2)$

- Causal Order \rightarrow FIFO order
- Fifo Order \nrightarrow Causal Order

Diffusion Causal

$$\text{broadcast}_p(m_1) \rightarrow \text{broadcast}_q(m_2) \Leftrightarrow \text{deliver}_p(m_1) \rightarrow \text{deliver}_q(m_2)$$

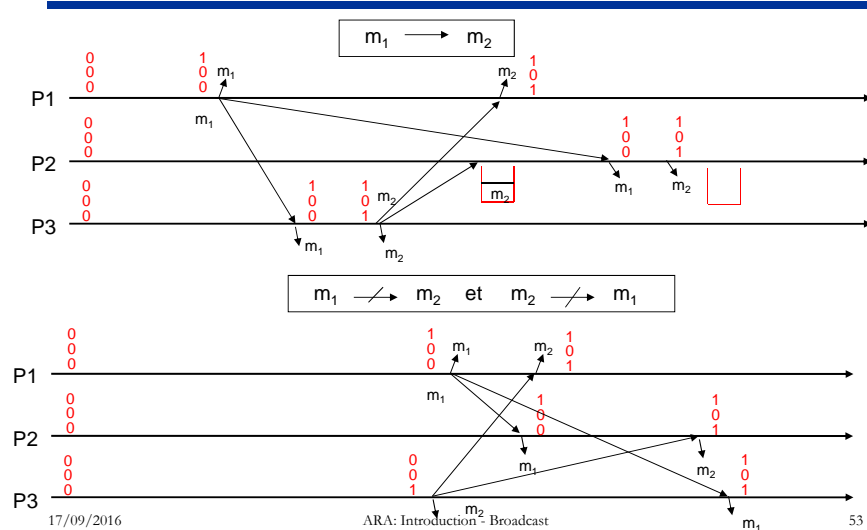


- Un algorithme de diffusion FIFO peut être transformé dans un algorithme de diffusion causal :

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

52

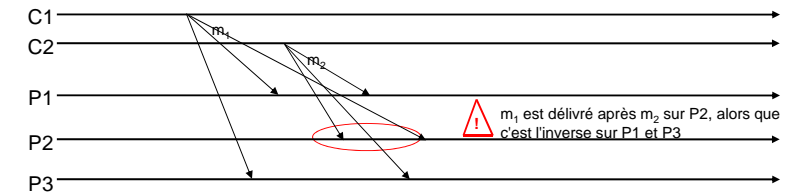
Diffusion Causal – algorithme avec horloges vectorielles - Exemple



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 \Rightarrow 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 le même ordre
- Ils décident la première valeur

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

Diffusion Atomique \Leftrightarrow Consensus

Chandra & Toueg 1996

Diffusion Atomique - ABCAST

■ Remarques :

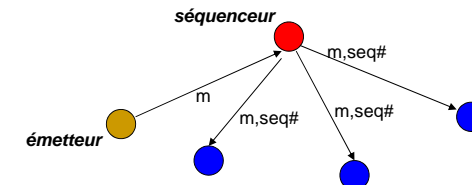
- ABCAST n'est pas réalisable dans un système asynchrone si on suppose l'existence de fautes (d'après FLP).
- ABCAST est réalisable (n nodes):
 - Avec un détecteur de pannes de classe P ou S en tolérant $n-1$ pannes
 - Avec détecteur de pannes de classe $\diamond S$ en tolérant $n/2 - 1$ pannes
 - Avec 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 = \emptyset ;

Émetteur :

OT_broadcast (m)
send m au séquenceur;

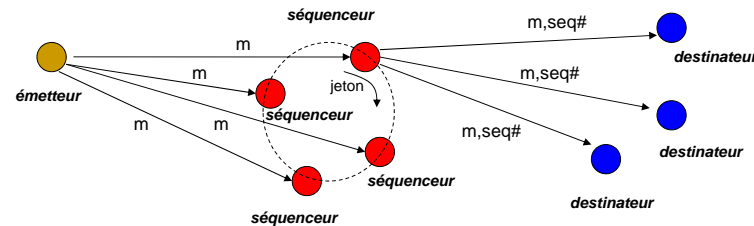
Séquenceur :

init :
seq#=1;
upon revc(m) do
send ($m, seq\#$) à tous processus
seq#++;

Destinateur :

upon revc(m) do
pend $\cup = \{m\}$
while ($\exists (m', seq\#) \in \text{pend} : seq\# = \text{nextdelv}$) 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 = ∅;
```

Émetteur :

```
OT_broadcast (m)
    send m à tous les séquenceurs;
```

Destinateur :

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

Séquenceur :

intit :

```
rec = ∅;
if (p = s1)
    token.seq# = 1
    token.liste = ∅;
```

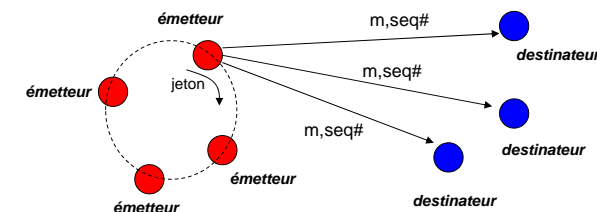
upon revc(m) do

```
rec ∪= {m}
```

upon revc(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 garder 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 = 0;
 send_pend = 0;

Destinateur :

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

Emetteur :

intit :
 send_pend = 0;
 if (p=s1)
 token.seq# = 1

procedure OT_broadcast (m)
 send_pend ∪= {m}

upon revc(token) do

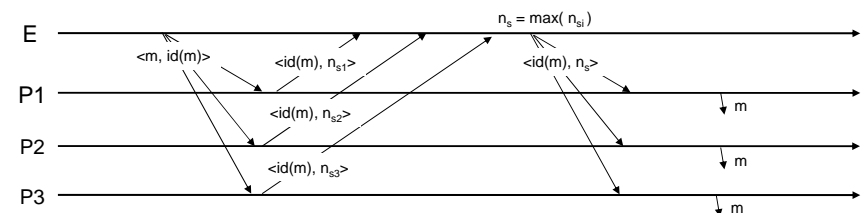
```
for each m' in send_pend do
    send (m',token.seq#) à tous les
    destinateurs
    token.seq#++;
    send_pend ∪= ∅;
    send (token) au prochain émetteur
```

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



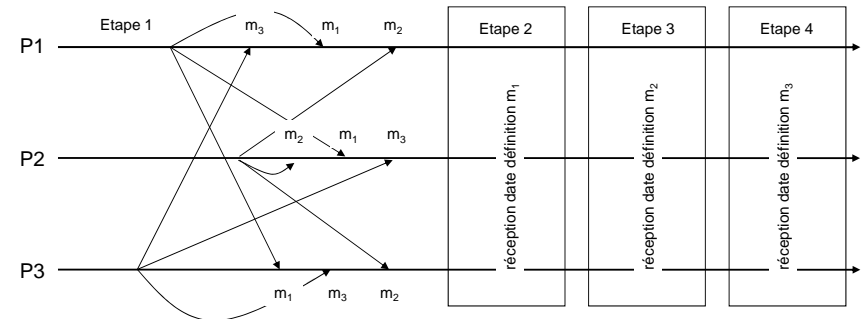
Les numéros proposés sont *<date logique réception, identité récepteur>* pour assurer un ordre total. Chaque processus maintient une file d'attente des messages en attente de numérotation définitive, triée de façon croissante sur les numéros.

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'étape 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																											
<table border="1"> <tr><td>m_3</td><td>m_1</td><td>m_2</td></tr> <tr><td>15.1</td><td>16.1</td><td>17.1</td></tr> <tr><td>N</td><td>N</td><td>N</td></tr> </table>	m_3	m_1	m_2	15.1	16.1	17.1	N	N	N	<table border="1"> <tr><td>m_2</td><td>m_1</td><td>m_3</td></tr> <tr><td>16.2</td><td>17.2</td><td>18.2</td></tr> <tr><td>N</td><td>N</td><td>N</td></tr> </table>	m_2	m_1	m_3	16.2	17.2	18.2	N	N	N	<table border="1"> <tr><td>m_1</td><td>m_3</td><td>m_2</td></tr> <tr><td>17.3</td><td>18.3</td><td>19.3</td></tr> <tr><td>N</td><td>N</td><td>N</td></tr> </table>	m_1	m_3	m_2	17.3	18.3	19.3	N	N	N
m_3	m_1	m_2																											
15.1	16.1	17.1																											
N	N	N																											
m_2	m_1	m_3																											
16.2	17.2	18.2																											
N	N	N																											
m_1	m_3	m_2																											
17.3	18.3	19.3																											
N	N	N																											

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


FA P1	FA P2	FA P3																											
<table border="1"> <tr><td>m_3</td><td>m_2</td><td>m_1</td></tr> <tr><td>15.1</td><td>17.1</td><td>17.3</td></tr> <tr><td>N</td><td>N</td><td>D</td></tr> </table>	m_3	m_2	m_1	15.1	17.1	17.3	N	N	D	<table border="1"> <tr><td>m_2</td><td>m_1</td><td>m_3</td></tr> <tr><td>16.2</td><td>17.3</td><td>18.2</td></tr> <tr><td>N</td><td>D</td><td>N</td></tr> </table>	m_2	m_1	m_3	16.2	17.3	18.2	N	D	N	<table border="1"> <tr><td>m_1</td><td>m_3</td><td>m_2</td></tr> <tr><td>17.3</td><td>18.3</td><td>19.3</td></tr> <tr><td>D</td><td>N</td><td>N</td></tr> </table>	m_1	m_3	m_2	17.3	18.3	19.3	D	N	N
m_3	m_2	m_1																											
15.1	17.1	17.3																											
N	N	D																											
m_2	m_1	m_3																											
16.2	17.3	18.2																											
N	D	N																											
m_1	m_3	m_2																											
17.3	18.3	19.3																											
D	N	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	m ₁	m ₂	m ₁	m ₃	m ₂	m ₃	m ₂
15.1	17.3	19.3	17.3	18.2	19.3	18.3	19.3
N	D	D	D	N	D	N	D


 m₁ est délivré sur P2

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

FA P1	FA P2	FA P3																					
<table><tr><td>m_1</td><td>m_3</td><td>m_2</td></tr><tr><td>17.3</td><td>18.3</td><td>19.3</td></tr><tr><td>D</td><td>D</td><td>D</td></tr></table>	m_1	m_3	m_2	17.3	18.3	19.3	D	D	D	<table><tr><td>m_3</td><td>m_2</td></tr><tr><td>18.3</td><td>19.3</td></tr><tr><td>D</td><td>D</td></tr></table>	m_3	m_2	18.3	19.3	D	D	<table><tr><td>m_3</td><td>m_2</td></tr><tr><td>18.3</td><td>19.3</td></tr><tr><td>D</td><td>D</td></tr></table>	m_3	m_2	18.3	19.3	D	D
m_1	m_3	m_2																					
17.3	18.3	19.3																					
D	D	D																					
m_3	m_2																						
18.3	19.3																						
D	D																						
m_3	m_2																						
18.3	19.3																						
D	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.

Broadcast and Consensus Building Blocks (briques de base)

Algorithmique répartie avancée - ARA
Master2

Luciana Arantes

Plan

- Consensus concept and properties
- Terminating Reliable Broadcast (*protocole de diffusion fiable avec terminaison*)
- View Synchronous Communication (*diffusion à vue synchrone*)
 - Group membership

Consensus

■ Specified by two primitives:

- *propose*
 - Processes exchange their proposal values
- *decide*
 - All correct processes decide on a single value through this primitive

Consensus Properties

1. Termination

- Every correct process eventually decides some value

2. Validity

- If a process decides v , then v was proposed by some process

3. Integrity

- No process decides twice

4. Agreement

- No two **correct** processes decide differently.

Uniform Consensus

1. Termination

- Every correct process eventually decides some value

2. Validity

- If a process decides v , then v was proposed by some process

3. Integrity

- No process decides twice

4. Agreement

- No two processes decide differently.

Terminating Reliable Broadcast

protocole de diffusion fiable avec terminaison

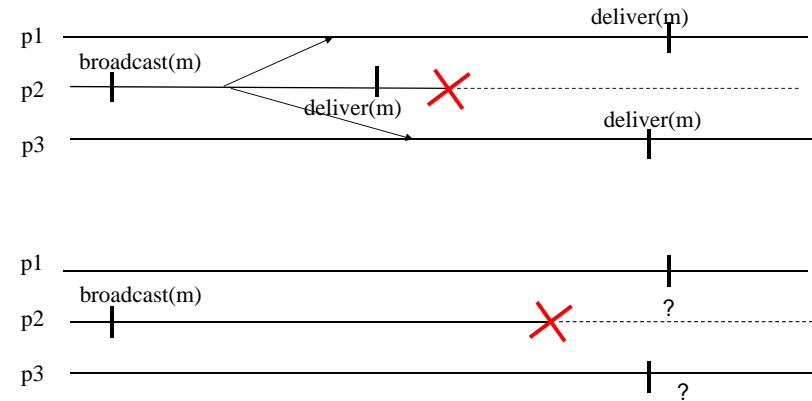
Terminating Reliable Broadcast

■ Motivation

- Consider that process p_i is known to have broadcast some message to all processes in the system.
 - Process p_i is an expected source of information and all processes must perform some specific task upon delivering p_i 's message . Thus, all processes wait then for p_i 's message.
- The use of a *uniform reliable broadcast* will ensure that if some process deliver m then all correct processes will deliver m .
 - A process can not decide if it should wait for m or not.
 - Impossible for a process p_j to distinguish the case where some process has delivered m (p_j should wait for m) and the case where no process will ever deliver m (p_j should not keep waiting for m).

Terminating Reliable Broadcast

(Uniform) Reliable broadcast



Terminating Reliable Broadcast

■ Terminating Reliable broadcast (TRB) is a (uniform) reliable broadcast with a specific termination property.

- Ensures precisely that every process p_j either delivers a message m broadcast by p_i or some indication F that m will never be delivered by any process.
 - The indication F is given in the form of a specific message, but it does not belong to the set of possible messages that processes broadcast.
- The TRB abstraction is a variant of consensus since all processes deliver the same message m or the message F .

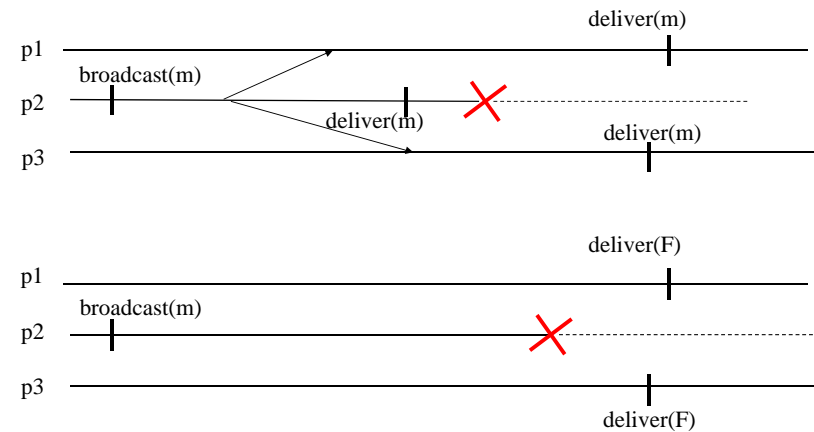
Terminating Reliable Broadcast

- Analogous to reliable broadcast, *terminating reliable broadcast (TRB)* is a communication primitive used to disseminate a message among a set of processes in a reliable way
- TRB is however strictly stronger than (uniform) reliable broadcast

Terminating Reliable Broadcast

- **src** = process that broadcasts the messages
- **Properties :**
 - *Validity*: If the sender *src* is correct and broadcasts *m*, then *src* eventually delivers *m*.
 - *Integrity* : If a correct process delivers a message *m* then either *m=F* or *m* was previously broadcast by *src*.
 - *(Uniform) Agreement* : For any message *m*, if a correct (any) process delivers *m*, then every correct process delivers *m*
 - *Termination*: Every correct process eventually delivers exactly one message.

Terminating Reliable Broadcast



Terminating Reliable Broadcast

- *Analogous to* reliable broadcast, correct processes in TRB agree on the set of messages they deliver
- *Analogous to* (uniform) reliable broadcast, every correct process in TRB delivers every message delivered by any process
- *Contrary to* reliable broadcast, every correct process delivers a message, even if the broadcast process *src* crashes

Terminating Reliable Broadcast

■ Algorithm

- Uses :
 - BestEffort_broadcast
 - Perfect Failure Detector *P* (synchronous system)
 - Consensus

Init :

`prop = ⊥;`

`correct = Π`

/ tous les processus */*

Terminating Reliable Broadcast

Consensus-based Algorithm

```

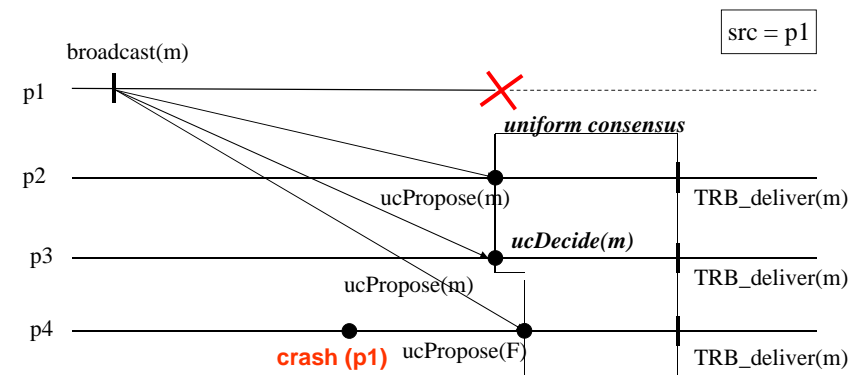
TRB_broadcast (m)                /* only called
by src */
  BestEffort_broadcast(m);
upon <crash | pj>
  correct = correct \ {pj}
upon <(src ∈ correct) and (prop = ⊥)>
  prop = F
  ucPropose <prop>
upon <(BestEffort_deliver (m)) and (prop =
vide)>
  prop = m
  ucPropose <prop>
upon <ucDecide, decision>
  TRB_deliver (decision)
  prop = ⊥
    
```

17/09/2016

ARA: Introduction - Broadcast

89

Terminating Reliable Broadcast



17/09/2016

ARA: Introduction - Broadcast

90

2. View Synchronous Communication

Diffusion à vue synchrone

17/09/2016

ARA: Introduction - Broadcast

91

View Synchronous Communication (Diffusion à vue synchrone)

- The view synchronous broadcast abstraction includes two abstractions :
 - Reliable broadcast + membership
- A *group membership (appartenance)* is a distributed service responsible for managing the membership of groups of processes.
 - The successive memberships of a group are called the *views*.
 - A group membership service maintains a consistent view among correct processes of all correctly functioning processes

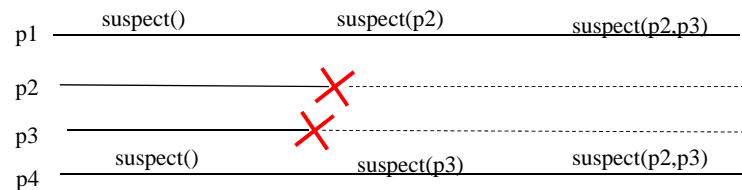
17/09/2016

ARA: Introduction - Broadcast

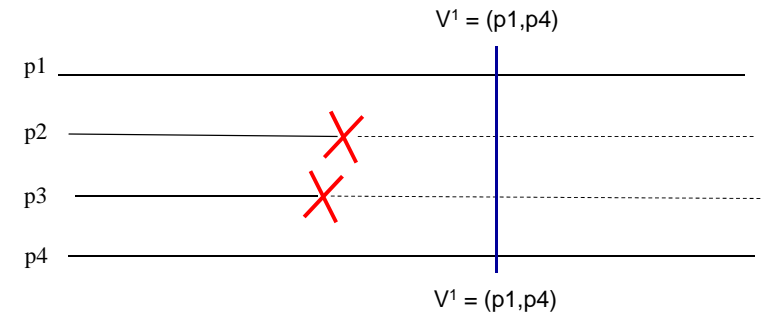
92

View Synchronous Communication (Diffusion à vue synchrone)

- In some distributed applications, processes need to know which processes are *participating* in the computation and which are not.
- Failure detectors provide such information. However, that information is *not coordinated* even if the failure detector is perfect.



Group membership



Group membership

■ A group membership service has two roles :

- provide to each member of the group a consistent view about the current composition of the group.
- coordinate the *join* of new members to the group and voluntarily *leave* of processes from the group.

■ Two models :

- *Primary partition group membership (primaire)*
 - One of the partition is recognized as the primary one and processes are allowed to deliver messages only if they belong to it.
 - Views are totally ordered.
- *Partitionable group membership (partition multiple)*
 - Allows all processes to deliver messages, regardless of the partition they belong to.
 - Views are partially ordered

Group membership

■ Group membership can be solved using consensus. However, due to FLP:

- Group membership **can not** be solved in asynchronous systems with crash failure.

■ To illustrate the group membership concept presentation, we just consider :

- *primary partition group membership*
- a group membership abstraction just to coordinate the information about *crashes*
 - the initial membership of the group is the complete set of processes, and subsequent membership changes are only caused by crashes.

Group membership

- Each view $V^i = (i, M)$ is a tuple where $i = V.id$ is a **unique view identifier** and $M = V.members$ is the **set of processes that belong to the view**.
 - We consider that initially V^0 includes all processes
 - $V^0 = (0, \Pi)$
 - All correct process deliver the same sequence of view
 - $V^0 = (0, M_0), V^1 = (1, M_1), V^2 = (2, M_2), \dots$
 - A process that delivers a view V^i is said to install view V^i
 - Process are informed about failures, having accurate knowledge about them. Contrary to failure detectors, information about failures are coordinated.

17/09/2016

ARA: Introduction - Broadcast

97

Group membership

■ Properties

- *Monotonicity* : If a process p installs view $V^j = (j, M_j)$ after installing $V^i = (i, M_i)$ then $j > i$ and $M_j \subset M_i$
- *Uniform Agreement* : If two processes install views $V^i = (i, M_i)$ and $V^{i'} = (i, M_{i'})$, then $M_i = M_{i'}$
- *Completeness* : If a process p crashes, then eventually every correct process installs $V^i = (i, M_i)$ with $p \notin M_i$.
- *Accuracy* : If some process installs a view $V^i = (i, M_i)$ and $q \notin M_i$, then q has crashed.

=> *Completeness* and *Accuracy* : perfect failure detector

17/09/2016

ARA: Introduction - Broadcast

98

Group membership

Consensus-based group membership algorithm

Uses :

Perfect failure detector P
Uniform Consensus

Init :

correct = Π
view = $(0, \Pi)$
wait = false.

upon <crash | p_j >

correct = correct $\setminus \{p_j\}$

upon <(correct \neq view.members) and (wait = false) >

wait = true
ucPropose <view.id+1, correct>

upon <ucDecide, (id, Members)>

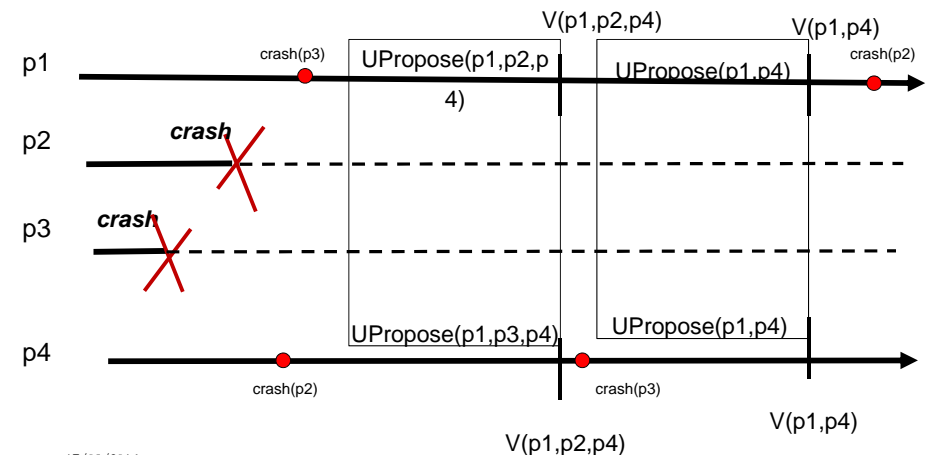
view = (id, Members);
wait = false
GroupMemb (view);

17/09/2016

ARA: Introduction - Broadcast

99

Group membership



17/09/2016

ARA: Introduction - Broadcast

100

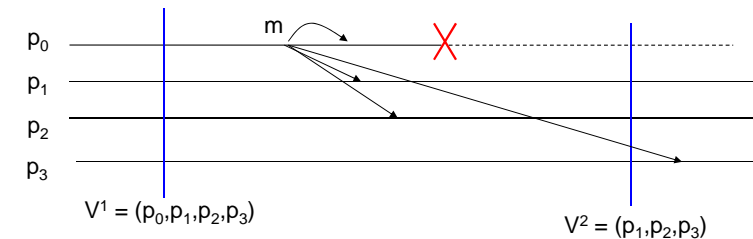
View Synchronous Communication (Diffusion à vue synchrone)

- **Goals : ensures both the reliability of the broadcast and consistency of the view information.**
 - The delivery of messages is coordinated with the installation of views
- **Motivation**
 - Suppose that :
 - Process q broadcast a message m and then crashes. The failure is detected by p_i and a new view is installed : $V = (i, M_i)$ such that $q \notin M_i$
 - Process p_j has delivered m , broadcast par q , before installing V .
 - View V is installed by all processes. Process p_i receives m . It would like then to discard m since it knows that q is crashed. However, it can not do it since process p_j has already delivered m .

➡ **Conflicting goals** : m has to be delivered by p_i to ensure the reliability of the broadcast and at the same time m should be discarded to ensure consistency with the view information.

View Synchronous Communication (Diffusion à vue synchrone)

Synchronous view is not guarantee

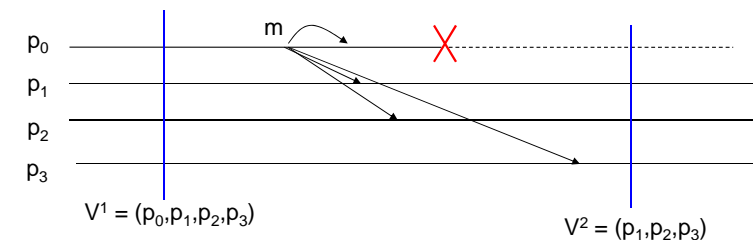


View Synchronous Communication (Diffusion à vue synchrone)

- **The installation of a new view is ordered with respect to the message flow.**
 - A process delivers (or broadcast) m in view V^i if the process delivers (or broadcast) m after installing V^i and before installing V^{i+1} (**View inclusion**).
 - "Gives the illusion" that failures occur at the same point in time with regard to message flow.

View Synchronous Communication (Diffusion à vue synchrone)

Synchronous view is guarantee



View Synchronous Broadcast

(Diffusion à vue synchrone)

■ Properties :

- Properties of *reliable broadcast*
- Properties of *group membership*
- *View inclusion* : If a process p delivers a message m from q in view V , then m was broadcast by q in view V .
 - A message m is delivered in the same view by different processes, and this view is the one where m was broadcast.
 - Addresses the problem of messages coming from processes already declared to have crashed.

View Synchronous Broadcast

(Diffusion à vue synchrone)

- If the application keeps *Vs_broadcasting* messages, the *view synchrony* abstraction might never be able to install a new view.
 - A specific event *block* is then used to request the application to stop broadcasting. New messages can be broadcast after the new view installation. The *block-ok* event is used by the application to acknowledge the block request.
 - When a process detects a crash, a "collective flush procedure of message delivery" must be executed before the installation of the new view.

Consensus-based View Synchronous Broadcast

■ Algorithm

- Uses
 - BestEffort_broadcast
 - Uniform Consensus
 - Perfect Failure Detector P
- Implement primitive
 - *VS_broadcast*
 - *VS_deliver*

Consensus-based View Synchronous Broadcast

■ Description

- Process p *VS_broadcast* message m by using the *BestEffort_broadcast* primitive. It adds then m to its *received* set (messages received by p).
- Whenever a message m , sent by q in the same view V , is delivered to p (*BestEffort_deliver*(m)), p adds m to its *received* set, and adds q to the set of processes that have acknowledged m (ack_m). Process p then broadcasts m , if it has not already done it.
- When all processes in the current view are in ack_m at process p , the message is *VS_deliver* by p .

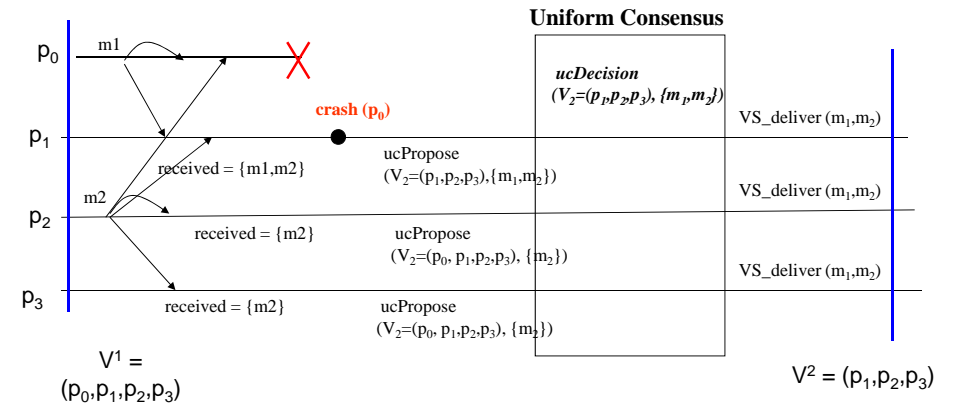
Consensus-based View Synchronous Broadcast

■ Description (cont.)

- If p detects a crash of one of the members of the current view, p initiates a "collective flush message delivery procedure"
 - It broadcasts the set of *received* messages.
 - As soon as p has collected the *received* set from every other process that p did not detect to have crashed, p proposes:
 - To consensus the *received* sets of all processes
 - A new view.
 - Upon de decision of the consensus
 - Each process first parses all *received* sets in the consensus decision and *VS_delivers* those messages p has not deliver yet.
 - Installs then the new view.

Consensus-based View Synchronous Broadcast

Execution Example



Bibliographie

- R. Guerraoui, L. Rodrigues. *Reliable Distributed Programming*, Springer, 2006
- K. Birman an T. Joseph . *Exploiting virtual synchrony in distributed systems*. Proceedings of the eleventh ACM Symposium on Operating systems principles, pages 123-138, 1987.

Epidemic Broadcast

Diffusion Epidémique

Epidemic Broadcast

- **The broadcast algorithms that we have seen till now are not scalable**
 - They consider a set of processes known by all processes from the beginning.
- **Epidemic algorithms are effective solution for disseminating in large scale and dynamic systems.**
 - They do not provide deterministic broadcast guarantees but just make probabilistic claims about such guarantees.
- **An epidemic broadcast uses a randomized approach where all the participants in the protocol should collaborate in the same manner to disseminate information.**

Epidemic Broadcast

- When a process p wishes to send a broadcast message, it selects k processes at random and sends the message to them
 - k is a typical configuration parameter called *fanout*.
- Upon receiving a message from p for the first time, a process q repeats the same procedure of p 's : q selects k gossip targets processes and forwards the message to them.
 - If a node receives the message twice, it simply discards the message
 - Each process needs to keep track of which messages it has already seen and delivered. The size of this buffer is also a scalable constraints.
- The step consisting of receiving a message and forwarding it is called a *round*.
 - An epidemic algorithm usually performs a *maximum number of rounds* r for each message.

Epidemic broadcast

- **Epidemic broadcast can only be applied to applications that do not require full reliability.**
 - The cost of full reliability is usually not acceptable in large scale systems.
 - However, it is possible to build scalable randomized epidemic algorithms which provide good reliability guarantees.
 - It exhibit a very stable behavior even in the presence of failures.

Epidemic Broadcast

- **Parameters associated with the configuration of gossip protocols :**
 - ***Fanout* (k):** number of nodes that are selected as gossip targets by a node for each message that is received by the first time.
 - Tradeoff associated between desired reliability level and redundancy level of the protocol.
 - ***Maximum rounds* (r):** maximum number of times a given gossip message is retransmitted by nodes.
 - Each message carries a *round value*, which is increased each time the message is retransmitted.
 - **Modes :**
 - *Unlimited mode*: the parameter maximum round is undefined
 - *Limited mode* : the parameter maximum round is defined with a value greater than 0.
 - Higher value: higher reliability as well as message redundancy.

Epidemic Broadcast

■ Probabilistic Broadcast

➤ Properties

- *Probabilistic validity* : There is a given probability such that for any two correct processes p_i and p_j , every message broadcast by p_i is eventually delivered by p_j with this probability.
- *No duplication* : No message is delivered more than once by a process
- *No creation* : If a message m is delivered by some process p_j , then m was previously broadcast by some process p_i .

Epidemic Broadcast

■ Strategies

- *Eager push approach* : Nodes send message to selected nodes as soon as they receive them for the first time
- *Pull approach* : Periodically, nodes query random selected nodes for information about recently received messages. When they receive information about a message they did not received yet, they explicitly request the message to their neighbors.
- *Lazy push approach* : When a node receives a message for the first time, it gossips only the message identifier. If a node receives a identifier of a message it has not received, it makes an explicitly pull request.
- *Hybrid approach* : First phase uses a push gossip to disseminate a message in best-effort manner. A second phase of pull gossip is used to recover messages not received in the first phase.

Eager Push Epidemic Broadcast

Algorithm

Init :
delivered = \emptyset

Epid_broadcast (m)
gossip(self, m, maxrounds);

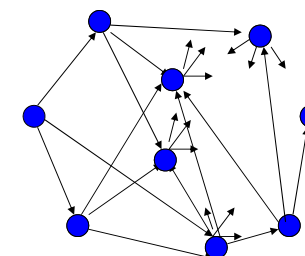
upon recv (pi, <src,m, r>)
if (m \notin delivered)
delivered = delivered \cup {m}
Epid_deliver(src,m)
if (r > 0)
gossip(self, m, maxrounds - 1);

Function chose-targets (ntargets)
targets = \emptyset
while (| targets| < ntargets) **do**
candidate = random (Π)
if ((candidate \notin targets) and
(candidate \neq self))
targets = targets \cup {candidate};
return targets

procedure gossip (src,msg,round)
for i \in chose-targets(fanout) **do**
send (i, msg, round);

Eager Push Epidemic Broadcast

Execution example



Fanout = 3 ; Maxround = 3

Epidemic Broadcast

- **Ideally, one would like to have each participant to select gossip targets at random from the entire system, as shown in the previous example.**
 - Realistic if it is deployed within a moderate sized cluster.
 - Such approach is not scalable :
 - High memory cost to maintain full membership information.
 - High cost of ensuring the update of such information.
- **Solution:**
 - Gossip-based (epidemic) broadcast protocols rely on *partial view*, instead of full membership information.

Epidemic Broadcast : Partial view

- **Partial view**
 - A process just knows a small subset of the entire system membership, from which it can select nodes to whom relay gossip messages
 - The membership protocol establishes *neighboring* association among nodes.
 - It must maintain the partial view at each node in face of dynamic changes in the system membership.
 - Joining of new nodes, crashes of nodes, etc.
 - A partial view must be a tradeoff between *scalability* against *reliability*
 - Small views scale better, while large views reduce the probability that processes become isolated or that network partitions occur.
 - **Overlay**
 - Partial views of all nodes of the system define a graph

Epidemic Broadcast : Partial view

- **Partial View Properties : related to the graph properties of the overlay defined by the partial view of all nodes**
 - *Connectivity* : the overlay should be connected : there should be at least one path from each node to all other nodes.
 - *Degree Distribution* : number of edges of the node.
 - *In-degree* of node n : number of nodes that have n in their partial view. It provides a measure of *reachability*.
 - *Out-degree* of node n : number of nodes in n 's view: measure of the importance of that node to maintain the overlay.
 - *Average Path Length* : the average of all shortest paths between all pair of nodes in the overlay.
 - *Accuracy* of node n : number of neighbors of n that have not failed divided by the total number of neighbors of n .

Epidemic Broadcast : Partial view

- **Strategies to maintain partial view**
 - *Reactive strategy* : a partial view only changes in response to some external event such as a joining of a node, a crash of a node, etc.
 - *Cyclic strategy* : A partial view is update every ΔT units of time, as a result of some periodic process that usually involves the exchange of information with one or more neighbors.
 - *Mixing strategy* : the partial view membership is included in the epidemic broadcast protocol
 - Whenever a process forwards a message, it also includes in it a set of processes it knows. Process that receives this message can update its own list of known processes.
 - It does not introduce extra communication to maintain membership.

Epidemic Broadcast : Partial view

■ Example : CYCLON

- Cyclic strategy : exchanging of view periodically among neighbors (*shuffling operation*), at a fixed period ΔT .
- A node keeps in cache pointers to its neighbors
 - Each pointer to a neighbor has a predictable lifetime
 - Field *age* : express the age of the pointer in ΔT intervals since the moment it was created.

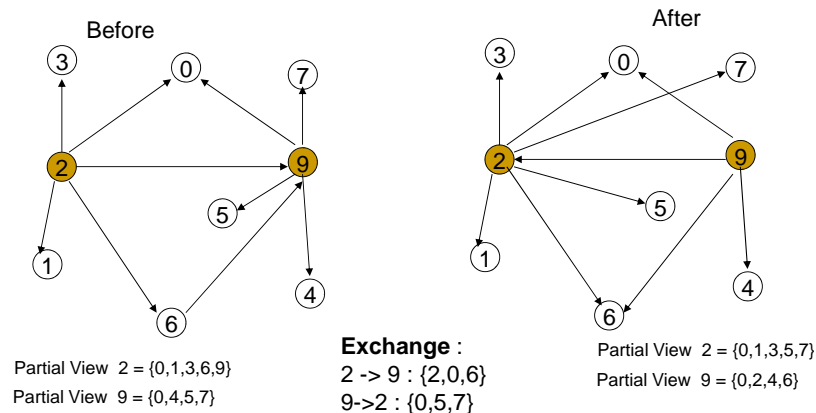
Epidemic Broadcast : Partial view

■ Example : CYCLON (cont.)

- Shuffling by node p :
 1. Increase by one the *age* of all its neighbors
 2. Select neighbor q with the highest age among all neighbors and l other random neighbors.
 3. Send the l random neighbors to q
 4. Upon receiving from q a subset of l of q 's neighbors :
 - discard those neighbors already in p 's cache
 - update p 's cache to include all remaining entries by firstly using empty caches slots (if any), and secondly replacing entries.

Epidemic Broadcast : Partial view

Cyclon : example execution



Gossip protocol in ad hoc Networks

■ An ad hoc network is a multi-hop wireless network with no fixed infrastructure

- Node broadcasts a message which is received by all nodes within one hop (neighbors)
- Gossiping protocol Gossip(p)[HHL06]
 - A source node sends the message m with probability 1.
 - Upon reception of m
 - first time,
 - it broadcasts m with probability p
 - it discards m with probability $1-p$
 - Otherwise it discards m

Gossip protocol in ad hoc Networks

- **If the source has few neighbors, chance that none of them will gossip and the algorithm dies.**
 - Solution : Gossip (p, k)
 - Gossip with probability 1 for the k hops before continuing to gossip with probability p .
 - Gossip (1,1) is equivalent to flooding.
 - Gossip ($p, 0$) : even the source gossips with probability p .

Bibliographie

- R. Guerraoui, L. Rodrigues. *Reliable Distributed Programming*, Springer, 2006 .
- J. C. A. Leitão. *Gossip-based broadcast protocols*. Master thesis's. 2007.
- P. Eugster, R. Guerraoui, A. Kermarrec and L. Massoulié. *From Epidemics to Distributed Computing*. IEEE Computer, 37, pages 60-67.
- K. Birman and T. Joseph . *Exploiting virtual synchrony in distributed systems*. Proceedings of the eleventh ACM Symposium on Operating systems principles, pages 123-138, 1987.
- S. Voulgaris, D. Gavidia, M. Sten. *Cyclon : Inexpensive membership management for unstructured p2p overlays*. Journal of Network and System Management. Vol 13, pages 197-217, 2005.
- Z.J. Haas, J. Halpern, L. Li. *Gossip-Based Ad Hoc Routing*. IEEE Transactions on Network, Vol. 14, N. 13, pages 479-491, 2006