

Programmation distribuée

Modèle d'un système distribué

On considère un système distribué qui possède les caractéristiques suivantes:

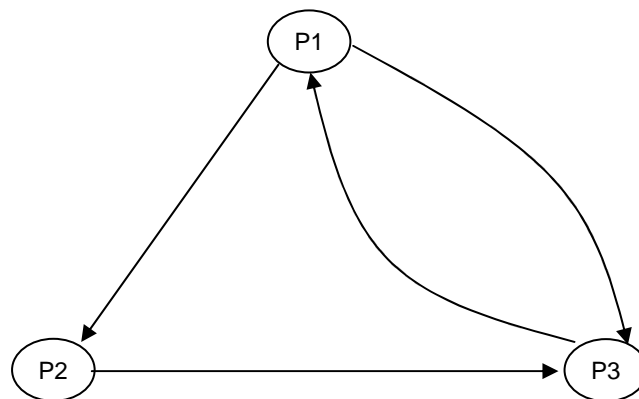
1. *Absence d'horloge commune.* Les processus s'exécutent sur des machines distantes qui possèdent des horloges locales. On ne peut pas assurer que toutes les horloges soient synchronisées.
2. *Absence de mémoire partagée.* Les processus n'ont pas accès à la totalité des données. Des algorithmes doivent être mis en œuvre pour leur permettre de connaître l'état global du système.
3. *Pas de détection de panne.* Le système que l'on considère est asynchrone. Dans ce contexte, on ne peut pas facilement faire la différence entre un processus lent et un processus en panne. Ce contexte complique les solutions des problèmes de consensus (impossibilité), élection,

Modèle

Les communications se font par envois de messages asynchrones et aucune hypothèse est faite sur les délais de transmissions.

Par contre, les transmissions sont sans erreurs et les messages ne sont jamais perdus (capacité infinie).

Dans ce contexte, un programme distribué se modélise comme un graphe orienté où les sommets représentent les processus et les arcs (orientés) les canaux de transmissions.

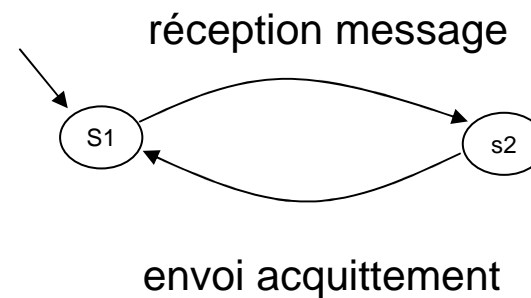
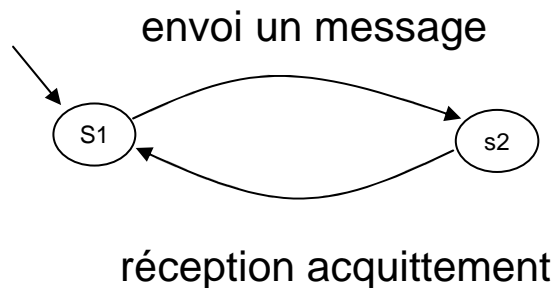


Processus

Un processus est défini par l'ensemble de ces états, une condition initiale (un sous-ensemble de l'ensemble des états) et un ensemble d'événements.

Un événement peut changer l'état d'un processus ainsi que l'état d'un canal de transmission.

Par exemple, on modélise un système client-serveur ou le serveur ne fait qu'acquitter les réceptions.



Horloge logique

Le but est de déterminer un ordre total sur les événements en possédant une horloge qui nous permet de déterminer l'ordre d'arrivée des événements (pensez aux algorithmes rencontrés qui utilisent une estampille temporelle).

D'après la discussion autour de la relation happen-before un ordre peut-être déterminé seulement pour les instructions intra-processus et pour les instructions relatives à l'émission/réception de message.

Une horloge logique C est une application de l'ensemble des événements dans les entiers naturels $C: E \rightarrow \mathbb{N}$ telle que

$$\forall e, f \in E \quad e \rightarrow f \Rightarrow C(e) < C(f)$$

Horloge logique

On peut aussi définir une horloge logique en associant une estampille temporelle aux états des processus.

$$\forall s, t \in S \quad s \rightarrow t \Rightarrow C(s) < C(t)$$

On impose en plus que la transmission d'un message ne prenne pas un temps nul.

Exemple: une horloge commune partagée par tous les processus est une horloge logique.

Horloge logique

```
public class LamportClock {  
    int c; // l'horloge logique  
    public LamportClock() { c = 1; }  
    public int getValue() { return c }  
    public void tick() {  
        c = c + 1; // explicitement exécutée après les événements internes  
    }  
    public void sendAction() {  
        // inclure l'horloge logique dans le message  
        c = c + 1;  
    }  
    public void receiveAction(int receiveValue) {  
        c = Util.max(c, receiveValue) + 1; // après réception d'un message l'horloge  
                                           // est mise-à-jour  
    }  
}
```

Horloge logique

On vérifie que l'algorithme satisfait

$$\forall s, t \in S \quad s \rightarrow t \Rightarrow s.c < t.c$$



l'horloge c dans l'état s

Pour définir un ordre total, on inclus l'identificateur de processus pour lever les ambiguïtés possibles

$$(s.c, s.p) < (t.c, t.p) \iff (s.c < t.c) \vee ((s.c = t.c) \wedge (s.p < t.p))$$



l'identificateur du processus qui se trouve dans l'état s

Vecteur d'horloge

L'horloge logique proposée n'assure pas que si $s.c < t.c$ alors $s \rightarrow t$.
En effet, (S, \rightarrow) définit un ordre partiel alors que l'ordre sur les entiers naturels est total.

Un vecteur d'horloge est une application $v : S \rightarrow N^k$ telle que

$$\forall s, t \in S \quad s \rightarrow t \iff s.v < t.v$$

 vecteur associé à l'état s

Pour comparer les vecteurs on doit définir un ordre partiel consistant avec la relation happen-before

Relation d'ordre

Soit deux vecteur x et y de dimension k on définit

$$x < y \iff (\forall 1 \leq i \leq k \quad x[i] \leq y[i]) \wedge (\exists 1 \leq j \leq k \quad x[j] < y[j])$$

$$x \leq y \iff (x < y) \vee (x = y)$$

L'ordre est partiel, $(2, 3, 0)$ et $(0, 4, 1)$ ne sont pas comparables.

Une horloge vectorielle associe à chaque état/événement un vecteur horloge.

Dans l'implémentation suivante, la taille du vecteur horloge est le nombre de processus

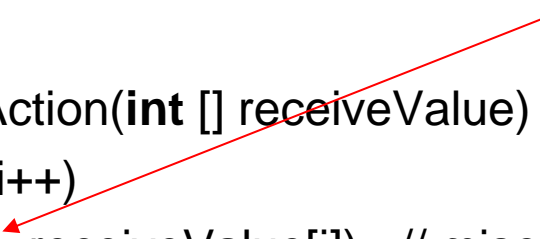
Horloge vectorielle

```
public class VectorClock {  
    public int [] v; // vecteur horloge  
    int myId;  
    int k; // nombre de processus  
    public VectorClock( int numProc, int id) {  
        myId = id;  
        k = numProc; // une entrée dans le vecteur par horloge  
        v = new int[numProc]; // le vecteur horloge  
        for(int i = 0; i < k; i++) v[i] = 0;  
        v[myId] = 1;  
    }  
    public void tick() {  
        v[myId]++; // après chaque événement/action on incrémente l'horloge  
    }  
}
```

Horloge vectorielle

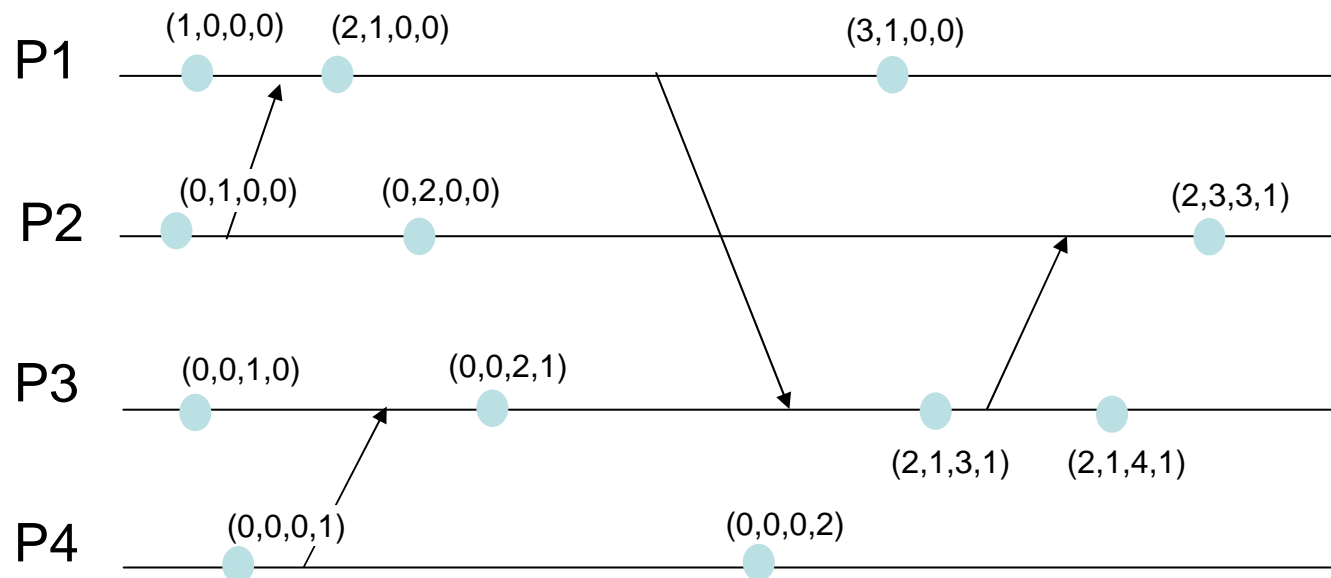
```
public void sendAction() {  
    // inclure le vecteur dans le message  
    v[myId]++; // on incrémente son horloge  
}  
public void receiveAction(int [] receiveValue) {  
    for( int i = 0; i < k; i++)  
        v[i] = Util.max(v[i], receiveValue[i]); // mise-à-jour de toutes les entrées  
    v[myId]++; // du vecteur horloge  
}  
public int getValue(int i) { return v[i]; }  
  
public String toString() { return Util.writeArray(v); }  
}
```

le vecteur local devient plus grand que le vecteur reçu



Horloge vectorielle

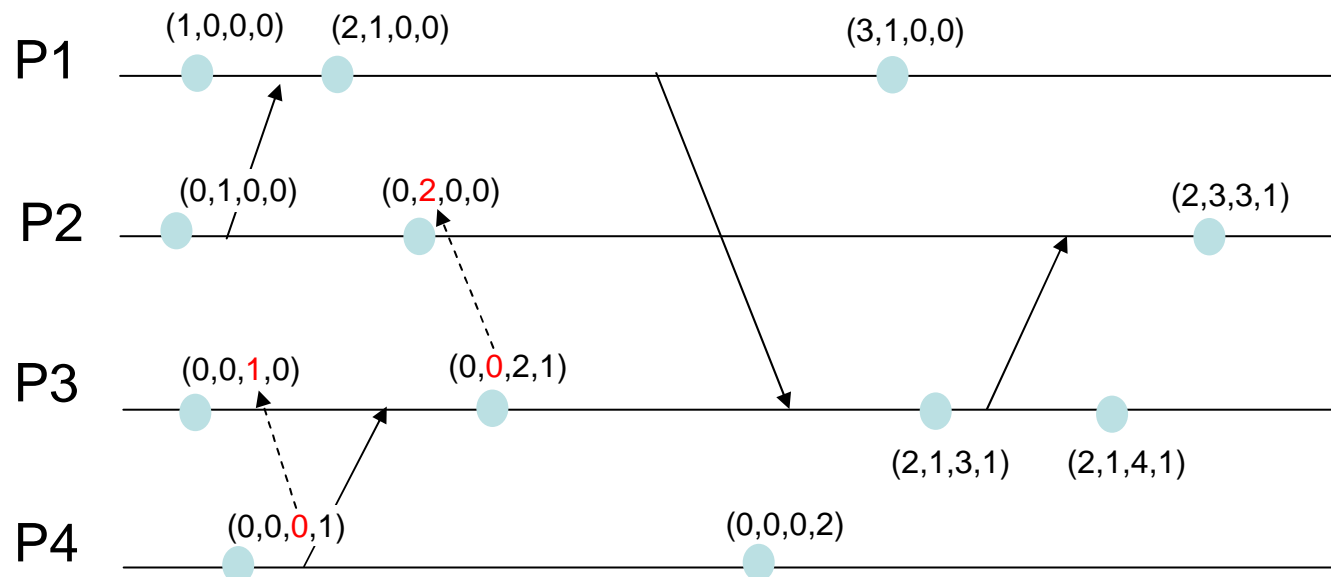
En pratique, c'est le linker qui va prendre en charge l'insertion du vecteur horloge dans le message et l'appel à la méthode *sendAction()*



Horloge vectorielle

On montre que

$$\text{si } s \neq t \text{ alors } s \not\rightarrow t \Rightarrow t.v[s.p] < s.v[s.p]$$



Horloge vectorielle

1^{er} cas: si $t.p = s.p$ (même processus) alors on a nécessairement $t \rightarrow s$
et donc $t.v[t.p] < s.v[s.p]$

2^{ème} cas: $t.p \neq s.p$

$s.v[s.p]$ est l'horloge du processus $P_{s.p}$, comme $s \not\rightarrow t$ alors le processus $P_{t.p}$ n'a pas connaissance de cette valeur (même pas par des processus intermédiaire sinon il y aurait une relation happen-before entre les événements). On a donc nécessairement

$$t.v[s.p] < s.v[s.p]$$

et on a montré $(s \not\rightarrow t) \Rightarrow \neg(s.v < t.v)$

Horloge vectorielle

Réciproquement, si $s \rightarrow t$ alors il existe une séquence de processus telle que $\mathbf{s} \rightarrow \mathbf{i} \rightarrow \mathbf{j} \dots \mathbf{k} \rightarrow \mathbf{t}$.

on a $\forall r \quad s.v[r] \leq i.v[r] \leq j.v[r] \leq \dots \leq t.v[r]$

De plus, on sait que $t \not\rightarrow s$, on a montré que cela implique $s.v[t.p] < t.v[t.p]$

On a donc bien

$$s \rightarrow t \Rightarrow s.v < t.v$$

Une horloge virtuelle permet donc aux processus de déterminer la relation happen-before en cours d'exécution.

Dépendance directe

L'horloge vectorielle nécessite d'échanger un vecteur de taille égale au nombre de processus. Pour limiter les échanges entre les processus on utilise **une horloge à dépendance directe** (*Direct Dependency Clock*). Cette horloge permet notamment de résoudre le problème de la section critique de manière distribuée.

Dans cet algorithme, les processus transmettent uniquement l'entrée de leur vecteur horloge qui correspond à leur identificateur. Cette horloge définit une relation de dépendance directe entre les processus, elle n'est pas transitive.

Dépendance directe

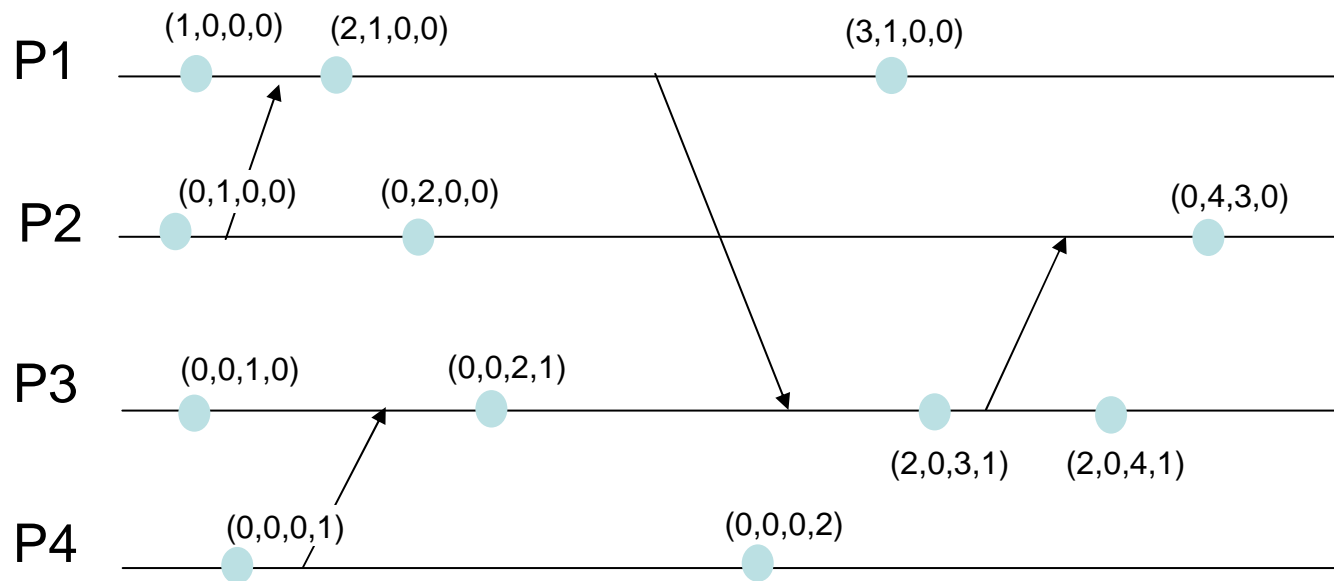
```
public class DirectClock {  
    public int [] clock;  
    int myId;  
    public DirectClock(int numProc, int id) {  
        myId = id;  
        clock = new int[numProc];  
        for( int i = 0; i < numProc; i++) clock[i] = 0;  
        clock[myId] = 1;  
    }  
    public int getValue(int i) { return clock[i]; }  
  
    public void tick() { clock[myId]++;}
```

Dépendance directe

```
public void sendAction() {  
    // inclure clock[myId] dans le message  
    tick();  
}  
  
public void receiveAction(int sender, int receiveValue) {  
    clock[sender] = Util.max(clock[sender], receiveValue);  
    clock[myId] = Util.max(clock[myId], receiveValue) + 1;  
}  
}
```

On remarque que l'algorithme est le même que celui d'une horloge logique i on ne considère que l'entrée du vecteur qui correspond au processus qui gère l'horloge.

Dépendance directe



Dépendance directe

On définit une nouvelle relation partiel d'ordre entre les événements/actions que l'on note \rightarrow_d (dépendance direct).

$s \rightarrow_d t$ s'il existe un chemin allant de s à t qui emprunte au plus un message dans le diagramme happen-before.

On peut montrer que cette relation satisfait la propriété suivante

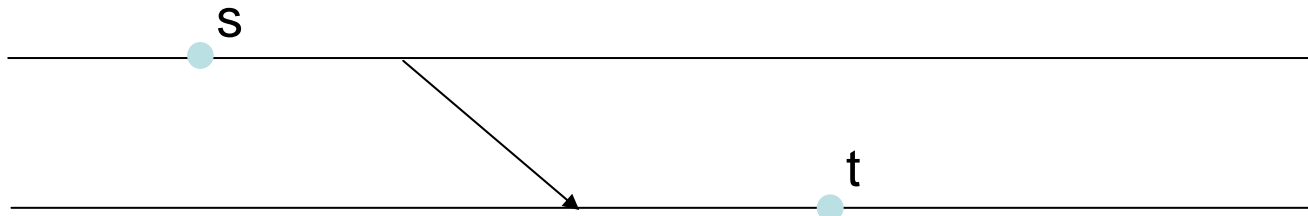
$$\forall s, t \quad s.p \neq t.p \quad (s \rightarrow_d t) \iff (s.v[s.p] \leq t.v[s.p])$$

Dépendance directe

‘preuve’: on montre d’abord que

$$s.p \neq t.p, \quad s \rightarrow_d t \Rightarrow s.v[s.p] \leq t.v[s.p]$$

La relation \rightarrow_d indique que depuis l’état s il existe un chemin menant à l’état t qui utilise au plus un lien correspondant à l’envoi d’un message



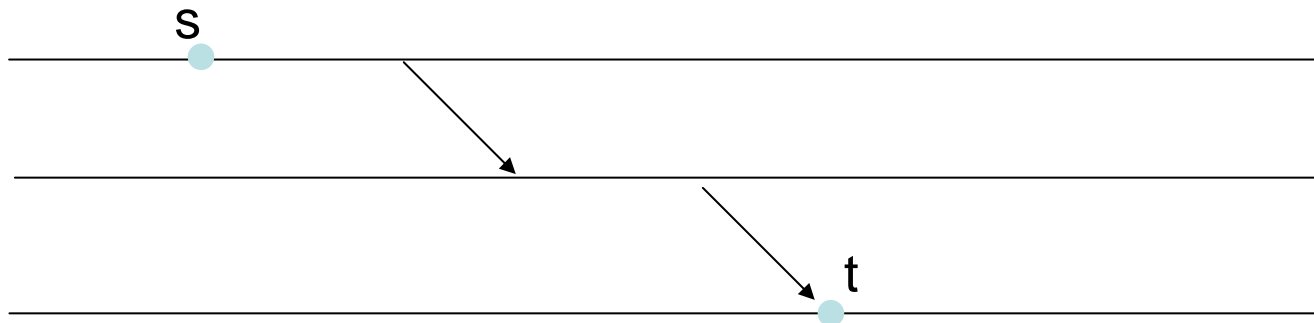
A la réception du message, le processus $t.p$ met à jour l’entrée de son horloge $[s.p]$ avec une valeur supérieur à $s.v[s.p]$.

Dépendance directe

‘preuve’: On montre la réciproque. Si $s \not\rightarrow_d t$, on a soit pas de chemin de s à t soit un chemin (minimal) qui emprunte au moins deux liens correspondant au transfert de message.

S’il n’y a pas de chemin, il est clair que $t.v[s.p] < s.v[s.p]$ puisque l’horloge $t.v[s.p]$ n’a pas été mise-à-jour (on a toujours $t.v[s.p] < s.v[s.p]$ sauf pour les états qui correspondent à la transmission-réception d’un message).

S’il y a un chemin indirect, le raisonnement est similaire puisque $t.v[s.p]$ n’est pas modifié



Application

On utilise les horloges à dépendance directe pour résoudre le problème de l'exclusion mutuelle dans un système distribué.

Chaque processus maintient une horloge pour estampiller les messages et une file d'attente pour mémoriser les requêtes d'accès à la section critique.

Cet algorithme (de Lamport) assure que les processus accèdent à la section critique dans l'ordre des estampilles temporelles de leurs requêtes.

Application

- Pour accéder à la section critique un processus envoie un message avec une estampille temporelle à tous les processus et ajoute une estampille à la queue.
- Lorsqu'un processus reçoit une requête, il l'a mémorise dans la queue avec son estampille temporelle.
- Pour libérer l'accès à la section critique un processus envoie un message à tous les processus.
- Lorsqu'un processus reçoit un message de libération de la section critique, la requête correspondante est supprimée de la file d'attente.
- Un processus peut accéder à la section critique si et seulement si
 - il a déposé une requête dans la file d'attente avec une estampille t
 - l'estampille t est plus petite que toutes les autres
 - il a reçu un message de tous les processus qui ont une requête dans la file d'attente avec une estampille plus grande que celle de sa requête (par exemple un acquittement)

Application

Pour implémenter l'algorithme on utilise deux tableaux:

- $q[j]$ qui contient l'estampille temporelle de la requête par le processus P_j
Si j n'a pas effectué de requête on utilise la valeur *Symbols.infinity*.

- $v[j]$ qui contient l'estampille temporelle du dernier message reçu du processus P_j . La composante $s.v[j]$ représente la valeur de l'horloge logique dans l'état s . La mise à jour se fait comme pour une horloge à dépendance directe, $v[j]$ et $v[myId]$ sont modifiés.

Plusieurs processus peuvent avoir la même estampille temporelle, on étend la relation d'ordre à relation totale en utilisant l'identificateur de processus. i entre en section critique si

$$\forall j, j \neq i \quad (q[i], i) < (v[j], j) \wedge (q[i], i) < (q[j], j)$$

Algorithme de Ricart et Agrawala

L'algorithme précédent assure l'exclusion mutuelle, utilise une horloge à dépendance directe et nécessite l'envoi de $3(N-1)$ messages ($N-1$ pour la requête, $N-1$ acquittement et $N-1$ libération).

L'algorithme de Ricart et Agrawala nécessite l'envoi de $2(N-1)$ messages. Il utilise un horloge de Lamport c'est-à-dire un entier qui est mis-à-jour à chaque réception de message.

Algorithme de Ricart et Agrawala

- Pour faire une requête un processus transmet un message estampillé *request*.
- Après réception d'une requête un processus transmet un message *okay* si il n'est pas intéressé par accéder la section critique ou si l'estampille de sa requête à une valeur plus grande. Sinon, l'identificateur du processus est mémorisé et le message *okay* sera transmis plus tard.
- Pour libérer la section critique un processus transmet un message *okay* à tous les processus.
- Un processus peut accéder la section critique si et seulement si il a effectué une requête et reçu un message d'acquiescement de la part de tous les autres processus

Algorithme de Ricart et Agrawala

```
int myts; // estampille temporelle locale
LamportClock c = new LamportClock(); // horloge logique
IntLinkedList pendingQ = new IntLinkedList(); // liste de proc. en attente
int numOkay; // nombre de message d'acquittement reçus
```

```
public RAMutex(...) {
    ...
    myts = Symbols.Infinity; // pas d'intérêt à accéder la section critique
}
```

```
public synchronized void requestCS() { // des processus indépendants invoquent
    requestCS et handleMsg
    c.tick(); myts = c.getValue(); broadcastMsg(' request ', myts); numOkay = 0;
    while (munOkay < N-1) myWait();
}
```

Algorithme de Ricart et Agrawala

```
public synchronized void releaseCS() {  
    myts = Symbols.Infinity;  
    while (! pendingQ.isEmpty()) {  
        int pid = pendingQ.removeHead();  
        sendMsg(pid, ' okay ', c.getValue());  
    }  
}  
  
public synchronized void handleMsg(Msg m, int src, String tag) {  
    int timeStamp = m.getMessageInt(); // récupération de l'estampille du message  
    c.receiveAction(timeStamp); // mis-à-jour de l'horloge max(c, timeStamp) + 1  
    if (tag.equals(' request')) {  
        if ((myts == Symbols.Infinity) || (timeStamp < myts) ||  
            ((timeStamp == myts) && (src < myId)))  
            sendMsg(src, ' okay ', c.getValue());  
        else pendingQ.add(src);  
    } else if (tag.equals(' okay ')) notify(); // voir requestCS()  
}
```

Algorithme de Ricart et Agrawala

On montre que cet algorithme assure l'exclusion mutuelle.

Supposons que les processus i et j se trouvent en section critique simultanément.

Ces processus ont déterminé une valeur $myts_i$ et $myts_j$ avant d'accéder à la section critique et envoyer cette valeur à tous les processus.

Cas 1: le processus j choisit $myts_j$ *après* avoir répondu au processus i (méthode `handleMsg` qui est synchronisée).

i choisit $myts_i \rightarrow i$ envoie la requête $\rightarrow j$ exécute `handleMsg` $\rightarrow j$ répond au processus i (*okay*) $\rightarrow j$ choisit $myts_j$

Avant de répondre le processus j met à jour son horloge et donc $myts_j > myts_i$. Le processus i ne répondra pas à la demande du processus j avant de sortir de la section critique et les deux processus ne peuvent pas s'y trouver simultanément.

Algorithme de Ricart et Agrawala

Cas 2: Le processus i choisit $myts_i$ après avoir répondu au processus j. Ce cas est similaire au premier.

Cas 3: Les processus i et j choisissent $myts_i$ et $myts_j$ avant de se répondre.

requestCS_i choisit $myts_i$ → envoi la requête
hMsg_i

requestCS_j choisit $myts_j$ → envoi la requête
hMsg_j

réception → réponse

réception → réponse

Les deux processus ne peuvent pas répondre les deux. En effet, les processus i et j répondent si $myts_i << myts_j$ et $myts_j << myts_i$.

Algorithme de Ricart et Agrawala

L'algorithme assure qu'un processus qui exécute requestCS entrera en section critique (pas d'insuffisance de ressource).

Le processus exécute requestCS_i, il détermine myts_i et envoie une requête à tous les processus. A un instant t, toutes les requêtes seront reçues et les processus mettront à jour leur horloge (c.receiveAction(...)).

Soit V l'ensemble des processus qui ont demandé à entrer en section critique avec une estampille temporelle inférieure ou égale à myts_i. Le cardinal de cet ensemble ne peut pas augmenter après le temps t.

L'ensemble des estampilles temporelles associés au processus de l'ensemble V sont ordonnées (ordre << total). La requête associée au processus possédant l'estampille la plus petite va être traitée par tous les processus qui vont y répondre par un message *okay*. Le cardinal V va donc diminuer. Par récurrence on montre que tous les processus vont accéder à la section critique dans l'ordre de leur estampille temporelle.

Autres stratégies pour l'exclusion mutuelle

Une stratégie classique pour résoudre le problème de l'exclusion mutuelle dans les systèmes distribués consiste à utiliser un jeton (token). Seul le processus qui dispose du jeton est habilité à accéder la section critique.

On peut implémenter cette idée en utilisant un algorithme **centralisé**.

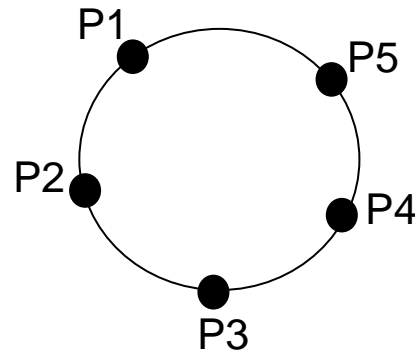
Un processus est désigné (comment?) pour être le coordinateur.

Chaque processus qui désire accéder la section critique envoie un message au coordinateur qui place le message dans une file FIFO.

Lorsqu'un processus sort de la section critique il informe le coordinateur qui envoie un message au processus prioritaire dans la file FIFO.

Autres stratégies pour l'exclusion mutuelle

Une autre implémentation consiste à définir un ordre cyclique sur les processus, la topologie devient celle d'un *anneau*.



Un message particulier, *le jeton*, circule de processus en processus. Seul le processus qui dispose du jeton peut accéder la section critique.

Le jeton peut aussi se déplacer seulement à la réception d'une requête.

Quorum

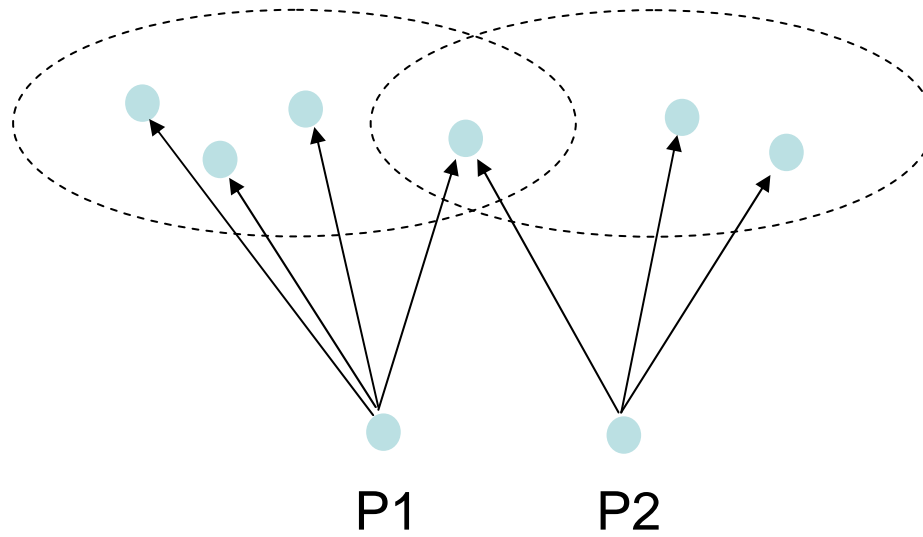
Les algorithmes qui utilisent un jeton sont vulnérables aux pannes du processus qui dispose du jeton.

Les algorithmes qui utilisent une estampille temporelle génèrent beaucoup de messages.

Une solution intermédiaire consiste à associer à chaque processus un sous-ensemble de processus, *request set*, auxquels le processus va demander la permission d'accéder la section critique.

Si ces sous-ensembles de processus ont des intersections non vides, on assure l'exclusion mutuelle. Typiquement on utilise des ensembles de $\lceil (N+1)/2 \rceil$ processus.

Quorum



Problème du consensus

Le problème du consensus dans les systèmes distribués s'énonce comme dans le cas de systèmes à mémoire partagée. Chaque processus possède une valeur initiale. Après exécution de l'algorithme tous les processus décident d'une valeur telle que

1. Tous les processus actifs (non en panne) décident de la même valeur (agreement).
2. Toutes les valeurs initiales peuvent être choisies (pour éviter la solution triviale) (validity)
3. Les processus actifs doivent décider d'une valeur en un temps fini (termination).

Dans l'énoncé des caractéristiques du protocole pour le problème du consensus, on distingue les processus actifs des processus qui peuvent être 'en panne'. On distingue plusieurs types de pannes.

Types de pannes

1. **Crash**, Un processus en panne arrête de s'exécuter et n'effectuera plus d'action dans le futur. Ce type de panne n'est pas détectable dans les systèmes asynchrones.
2. **Crash + lien**, soit un processus arrête de s'exécuter soit un lien de communication devient inutilisable. On distingue deux sous-cas, dans le premier les processus restent connectés dans le second ils deviennent déconnectés.
3. **Omission**, un processus omet de transmettre ou de recevoir un sous-ensemble de message qu'il aurait du recevoir/transmettre en exécutant le protocole.
4. **Byzantine**, un processus à un comportement imprévisible, des messages peuvent être transmis avec des valeurs fausses, différents messages peuvent être transmis à différents processus, ...

Systèmes synchrones

Un résultat de Fischer, Lynch et Paterson (FLP) montre que le **problème du consensus qui tolère la panne (crash) d'un processus ne peut pas être résolu dans un système asynchrone**. La démonstration de ce résultat est 'assez' similaire à celle que l'on a vu dans les systèmes à mémoire partagée.

Par contre, on peut résoudre ce problème dans les systèmes synchrones si on peut borner le nombre de processus susceptibles de tomber en panne.

Rappel: Un système est synchrone si on peut borner le temps qui sépare l'émission d'un message de sa réception.

Consensus

Chaque processus exécute le protocole suivant

P_i ::

var

V : ensemble des valeurs reçues, initialement $\{v_i\}$

for $k:=1$ to $f+1$ do

transmettre à tous les processus

$\{v \in V \text{ telles que } P_i \text{ na pas encore transmis } v\}$

recevoir S_j de tous les processus $P_j, j \neq i$

end

choisir $y = \min(V)$

Consensus

Comme le système est supposé synchrone, la réception de tous les messages utilise un temporisateur. Après avoir envoyé les messages, il attend un temps fini la réponse des autres processus.

Le protocole se termine donc en un temps fini (termination).

le protocole est valide car la valeur choisie appartient à l'ensemble V .

Le protocole supporte au plus la panne de f processus.

En effet, soit V_i l'ensemble des valeurs reçues par le processus P_i après les $f+1$ transmissions/réceptions.

On montre que si $x \in V_i$ alors $x \in V_j$ pour P_j un processus correct, c'est-à-dire qui a exécuté le protocole jusqu'à la fin.

Consensus

Cas 1: on suppose que la valeur x à été ajoutée à l'ensemble V_i au tour $k < f + 1$. Dans cette situation, le processus à transmis la valeur à tous les processus corrects.

Cas 2: la valeur à été ajoutée au tour $k = f + 1$

Il existe donc un processus qui à reçu cette valeur au tour f et transmise

Il existe donc un processus qui à reçu cette valeur au tour $f-1$ et transmise

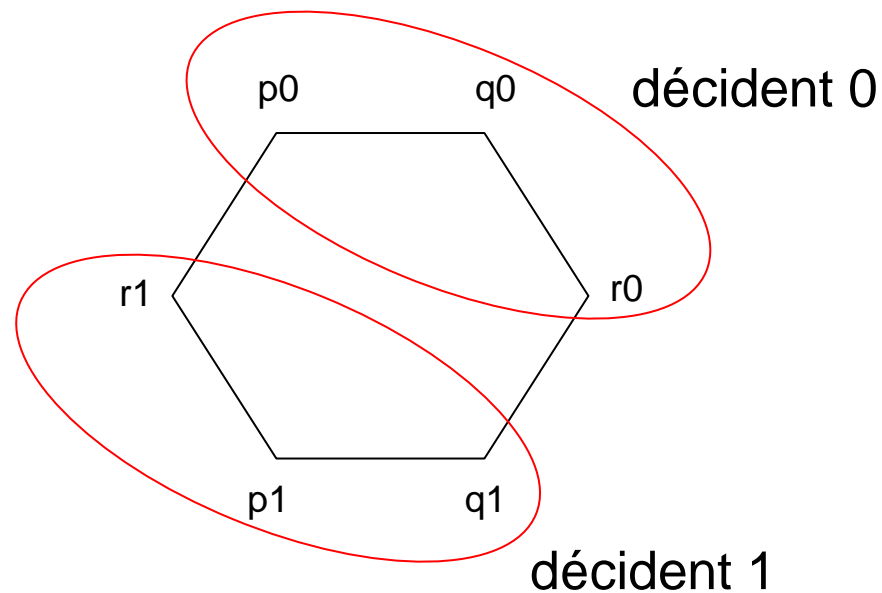
...

Il existe donc $f+1$ processus distincts qui on reçus cette valeur et l'ont transmise. Par hypothèse le nombre de processus qui tombe en panne est plus petit que $f+1$, il y a donc un processus correct qui à reçu cette valeur et l'a transmise à tous les processus (corrects).

Erreur Byzantine

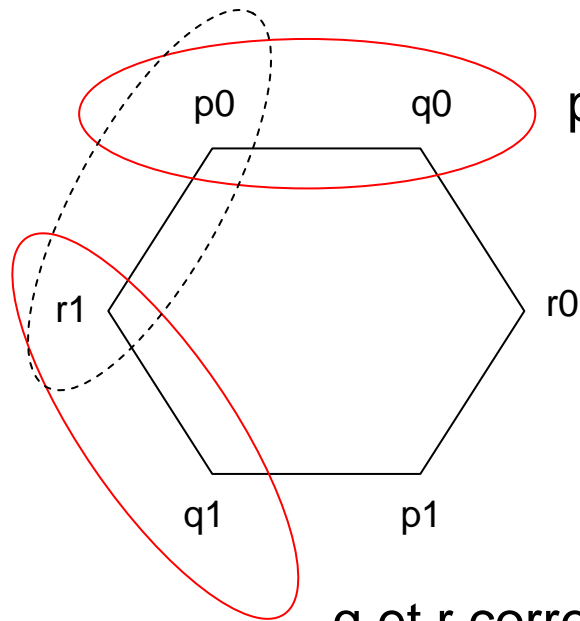
On considère le problème du consensus en tolérant des erreurs de type Byzantine de la part des processus.

Avec trois processus, le problème ne peut pas tolérer de panne.



Erreur Byzantine

quelle valeur choisir? dépend de q qui peut transmettre 0 à p et 1 à r



p et q correctes, décident 0

q et r correctes, décident 1

De manière générale, on ne peut pas résoudre ce problème si $n \leq 3f$

Erreur Byzantine

On considère le problème du consensus comme un problème de décision. A un ensemble de valeurs correspondant aux valeurs annoncées par les processus, on doit associer une valeur.

$$0\ 0\ 0 \rightarrow 0$$

$$1\ 1\ 1 \rightarrow 1$$

$$0\ 0\ * \rightarrow 0 \text{ pour tolérer une défaillance de } r$$

$$*\ 1\ 1 \rightarrow 1 \text{ pour tolérer une défaillance de } p$$

Quelle valeur associer à $0\ *\ 1$?

$0\ *\ 1 \rightarrow 0$ et la valeur reçue par q est 1 ce n'est pas cohérent avec $0\ 1\ 1 \rightarrow 1$

$0\ *\ 1 \rightarrow 1$ et la valeur reçue par q est 0 ce n'est pas cohérent avec $0\ 0\ 1 \rightarrow 0$

Coordinateur tournant

On peut résoudre ce problème si on suppose que le nombre de processus défaillant f satisfait $N > 4f$.

L'algorithme procède à $f+1$ étapes.

A chaque étape un processus est désigné comme le coordinateur.

Chaque processus mémorise une valeur de préférence qui est initialement sa valeur initiale.

Pendant une étape, les processus échangent leur valeur de préférence. Chaque processus choisit comme valeur de préférence la valeur qu'il a reçue le plus de fois -> mise à jour de *myValue*

Ensuite, chaque processus reçoit la (nouvelle) valeur du coordinateur *kingValue*.

Coordinateur tournant

Les processus peuvent ne pas recevoir la valeur du coordinateur si ce dernier est défaillant. Dans ce cas, ils choisissent une valeur par défaut.

Si le nombre de valeur reçues qui déterminent $myValue \leq N/2 + f$ alors le processus exécute $myValue = kingValue$. Sinon, la valeur n'est pas modifiée.

Code partiel

```
public synchronized void handleMsg(Msg m, int src, String tag) {  
    if (tag.equals(« phase 1 »)) V[src] = m.getMessageInt();  
    else if (tag.equals(« king »)) kingValue = m.getMessageInt();  
}
```

// exécutée en premier pour le consensus

```
public synchronized void propose (int val) {  
    for(int i = 0; i < N; i++) V[i] = defaultValue;  
    V[myId] = val;  
}
```

Code partiel

```
public int decide() {  
    for(int k = 0; k<=f; k++) { // f+1 étapes  
        broadcastMsg(« phase 1 », V[myId]);  
        Util.mySleep(Symbols.roundTime); // réseau synchrone  
        synchronized(this) {  
            myValue = getMajority(V);  
            if (k == myId) broadcast(« king », myValue);  
        }  
        Util.mySleep(Symbols.roundTime);  
        synchronized(this) {  
            if (numCopies(V, myValue) > N/2 + f) V[myId] = myValue;  
            else V[myId] = kingValue;  
        }  
    }  
}
```

Analyse de l'algorithme

Supposons qu'il existe une étape pendant laquelle tous les processus corrects choisissent la même valeur. Cette valeur persiste jusqu'à la fin de l'exécution du protocole. En effet,

$$N > 4f \Leftrightarrow N - N/2 > 2f \Leftrightarrow N - f > N/2 + f$$

Le nombre de valeur échangée par le processus corrects est suffisamment grand pour que la valeur *kingValue* ne soit jamais considérée.

Comme le nombre d'étape est $f+1$, pendant au moins une étape le coordinateur est un processus correct. Durant l'étape correspondante un processus choisit *myValue* si et seulement si il a reçu au moins $N/2 + f + 1$ telle valeur et le coordinateur choisit aussi cette valeur (au moins $N/2 + 1$ réception) ainsi que tous les processus correctes.

Transactions

Une transaction est composée d'un ensemble d'actions (d'opérations) telles que la séquence apparait comme une action indivisible.

Pour un observateur, la transaction apparait comme exécutée ou non (l'ensembles de opérations sont exécutées ou aucune opération est exécutée).

Plusieurs transactions peuvent s'effectuer de manière concurrentes et indivisibles.

Lorsqu'une transaction est réalisée, le résultat de ses opérations est permanent.

Exemple

On considère une transaction qui consiste à transférer une somme d'argent d'un compte A vers un compte B.

La transaction se décompose:

1. débiter le compte A
2. transférer l'argent vers le gestionnaire du compte B (message).
3. créditer le compte B

compte A

800

600

compte B

600

800

Une transaction T2 qui retourne la somme des comptes A et B retourne toujours 1400.

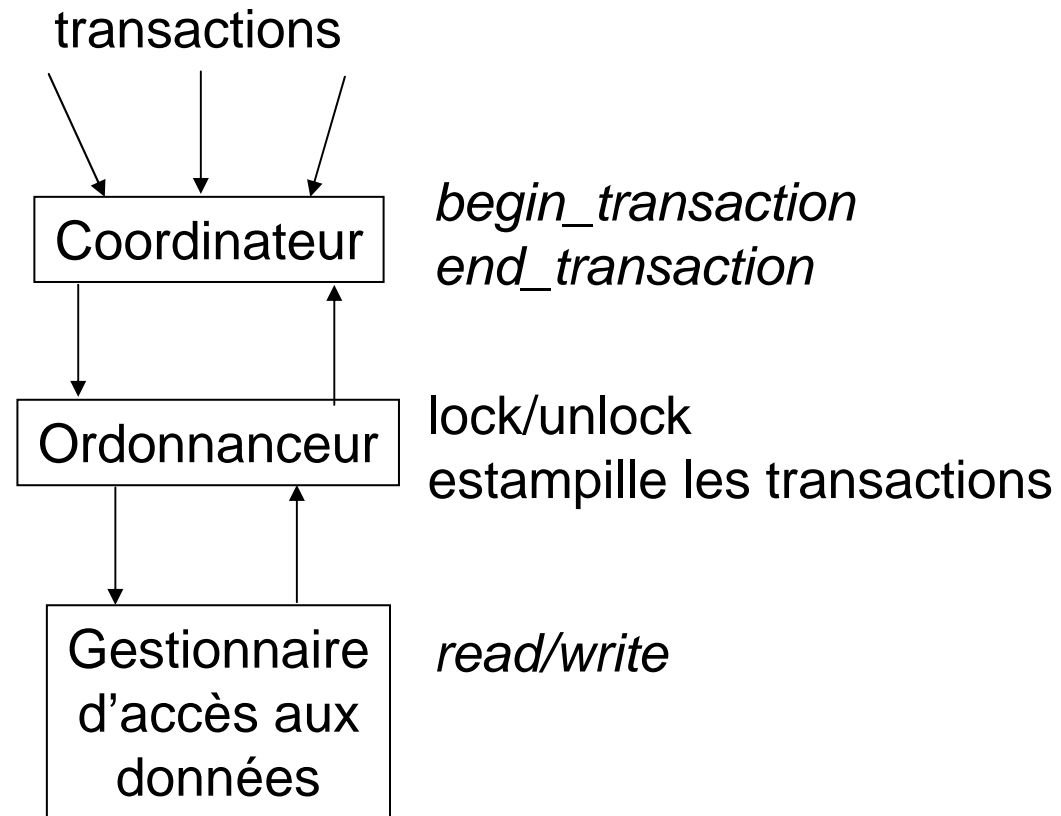
Exemple

Si une erreur survient entre le moment où le compte A a été débité et le moment où le compte B a été débité la transaction n'est pas *confirmée* (committed), le compte A n'est pas modifié.

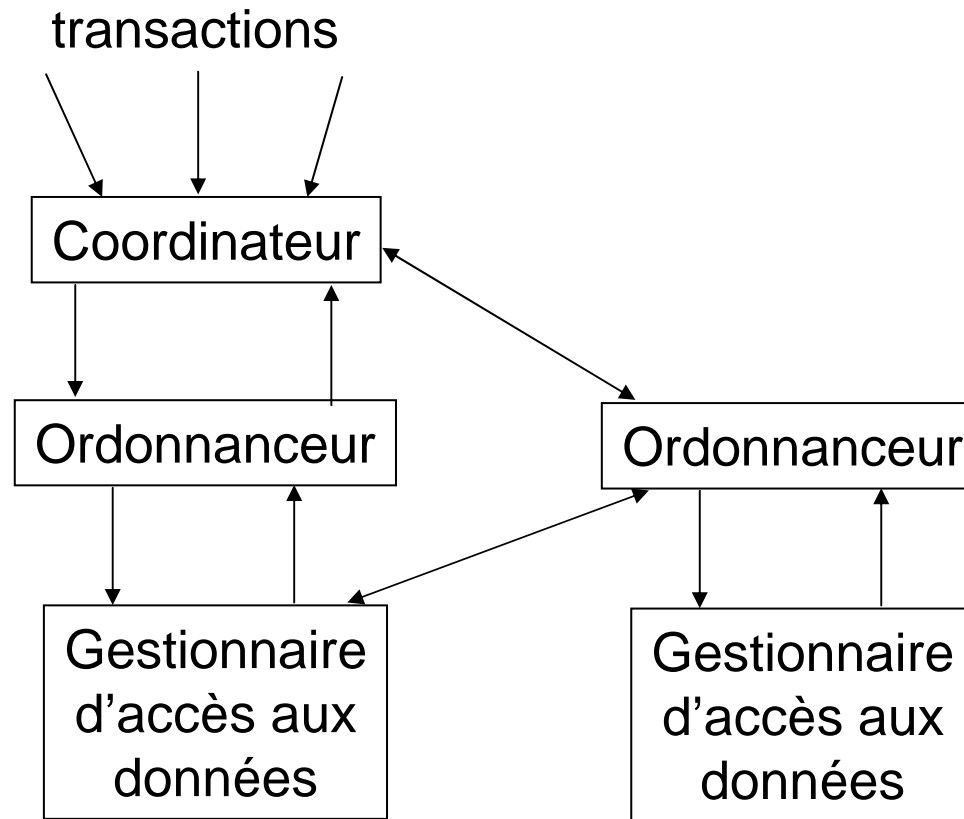
Une transaction (sur un objet) doit implémenter les méthodes suivantes:

1. *begin_transaction*
2. *end_transaction*
3. *abort_transaction*
4. *read*
5. *write*

Organisation



Organisation



Propriétés

Les propriétés qui doivent être assurées par une transaction sont:

1. **Atomicity**: la transaction apparaît comme une opération atomique. D'autres processus ne peuvent pas 'voir' les états intermédiaires.
2. **Consistency**: Une transaction doit respecter les contraintes d'intégrité du système. Par exemple, le transfert d'argent entre deux compte doit préserver la somme totale d'argent dans le système. La consistance est définie par des invariants.
3. **Isolation**: une transaction est isolée de l'effet d'une autre transaction, il n'y a pas d'interférence entre les transactions. L'effet de plusieurs transactions concurrentes est identique à l'exécution des transaction séquentiellement (sérialisable)
4. **Durability**: Une fois qu'une transaction est confirmée son effet est permanent.

On se réfère à ces propriétés en utilisant l'acronyme ACID.

Exemple

On considère trois compte A, B et C, chaque compte est crédité de 1000.-. On désire créditer 100.- de A vers B, 200.- de B vers C. Le solde final des trois compte doit être de 900.-, 900.- et 1200.-

Transaction A
begin_transaction
x = read(A) 1
x = x - 100 2
write(x,A) 3
x = read(B) 4
x = x + 100 11
write(x,B) 12
end_transaction

Transaction B
begin_transaction
y = read(B) 5
y = y - 200 6
write(y,B) 7
y = read(C) 8
y = y + 200 9
write(y,C) 10
end_transaction

L'exécution en rouge donne les soldes 900.-, 1100.-, 1200.-, qui est incorrecte

Exemple

Pour assurer l'atomicité on peut avant d'exécuter une transaction bloqué l'accès à la base de donnée. Ainsi, toutes les exécutions sont réellement séquentielles.

Une meilleure technique consiste à décomposer la transaction en deux phases. Une première pendant laquelle on bloque l'accès aux comptes et une deuxième phase pendant laquelle on débloque les accès.

Les verrous peuvent être associé à une donnée particulière, plus généralement on associe un verrou à toutes les données qui satisfont un prédicat logique (valeur inférieur à ...)

Example – two-phase locking

Transaction A
begin_transaction
lock(A)
x = read(A)
x = x - 100
write(x,A)
lock(B)
x = read(B)
x = x + 100
write(x,B)
unlock(A)
unlock(B)
end_transaction

Ordonnanceur – two-phase locking

L'ordonnanceur reçoit des requêtes correspondants à des opérations à effectuer dans la base de donnée $oper(T,x)$, T est la transaction et x la donnée.

1. Si l'opération est en conflit avec une autre opération la requête est temporisée. Sinon, un verrou sur la donnée x est donné à la transaction T et l'opération est transmise au gestionnaire d'accès des données
2. L'ordonnanceur libère le verrou seulement lorsque le gestionnaire l'a informé que l'opération est terminée
3. Une fois que l'ordonnanceur a libéré un verrou associé à une transaction il ne va plus jamais donner de verrou à cette transaction.

Attribuer les verrous selon cette politique assure que les données accédées sont toujours cohérentes.

Ordonnanceur – estampilles temporelles

Une approche différente consiste à utiliser une horloge logique. A chaque transaction est associé une *estampille temporelle* $ts(T)$ et chaque opération de T est estampillée avec $ts(T)$.

A chaque donnée sont associés deux estampilles temporelles $tsRD(x)$ qui est la date de la dernière lecture et $tsWR(x)$ qui est la date de la dernière écriture.

Lorsque l'ordonnanceur considère une opération $read(T, x)$, il l'exécute seulement si $ts(T) > tsWR(x)$ et exécute $tsRD(x) = \max(ts(T), tsRD(X))$.

Si $ts(T) < tsWR(x)$ une transaction qui a commencé après T a modifié x et la transaction T est annulée (*abort*). L

Ordonnanceur – estampilles temporelles

De même, si l'ordonnanceur considère une opération $write(T, x)$, il l'exécute seulement si $ts(T) > tsRD(X)$. Si $ts(T) < tsRD(X)$ la transaction T est annulée (*abort*) car la donnée x a été lue par une transaction plus récente que T .

En pratique, les estampilles temporelles permettent de limiter le temps de gestion des verrous. De plus, cette politique ne peut pas conduire à des interblocages.

Récupérer les erreurs

Pour récupérer l'état de la base de données après d'éventuels erreurs on considère deux techniques courantes:

Espace de travail privé (private workspace): Une transaction n'affecte pas les données originales. Les objets modifiés pas la transaction sont copiés.

Si la transaction est interrompue (abort) ces copies sont détruites.

Si la transaction est confirmée, ces copies deviennent les originaux.

journal des opérations (logging): Les mise-à-jour sont effectuées directement dans la base de données. Un journal des écritures est maintenu. Si la transaction doit être interrompue on peut remettre la base de données dans l'état original.

Confirmation d'une transaction

Lorsque plusieurs sites sont impliqués dans une même transaction ils doivent tous décider d'annuler la transaction ou de la confirmer.

L'algorithme utilisé doit être tolérant aux pannes et satisfaire:

1. Tous les processus corrects prennent la même décision
2. Si un processus décide d'annuler la transaction, tous les processus doivent l'annuler.
3. S'il n'y a pas de pannes, tous les processus doivent prendre une décision.
4. Les processus corrects doivent prendre une décision.

On considère l'algorithme suivant qui suppose que les liens de communication sont fiables et satisfait les trois premières contraintes.

Confirmation distribuée

1. Le coordinateur envoie un message *request* à tous les sites
2. Chaque site répond avec un message *yes* pour signifier au coordinateur qu'il peut confirmer la transaction, *no* dans le cas contraire
3. Le coordinateur attend les réponses de tous les sites, s'il reçoit au moins un message *no* il envoie un message *finalAbort*, sinon il confirme la transaction en envoyant un message *finalCommit*.
4. Les sites exécutent l'action correspondante.

Un site peut tomber en panne après avoir transmis un message *yes*. Dans cette situation, après remise en route (le canal de transmission est fiable) il doit être capable de confirmer la transaction.

Confirmation distribuée

Si un site ne reçoit pas de message *request* (il utilise un temporisateur) il annule la transaction et transmet un message *no* au coordinateur.

Le coordinateur peut aussi tomber en panne après avoir transmis un message *request*. Dans cette situation, un site qui a transmis un message *yes* doit contacter tous les sites impliqués dans la transactions et ils doivent se mettre d'accord (consensus) si oui ou non la transaction peut être confirmée. Si au moins un site a reçu un message *no*, la transaction doit être annulée.

Instantané global

Certaines applications requièrent d'obtenir une vision globale à un instant donné de l'état d'un système distribué. Par exemple, pour réaliser un inventaire.

Pour connaître l'état global courant du système, le système doit cesser de fonctionner.

Une autre stratégie consiste à obtenir un état global du système à un instant dans le *passé*. Par exemple, pour déterminer des propriétés qui sont stables (une fois réalisées elles le sont de manière permanente) il est suffisant d'obtenir des informations du système à des instants passés (déterminer s'il y a interblocage, perte du jeton, ...)

Un tel algorithme réalise un instantané global du système (*global snapshot*).

Instantanée global

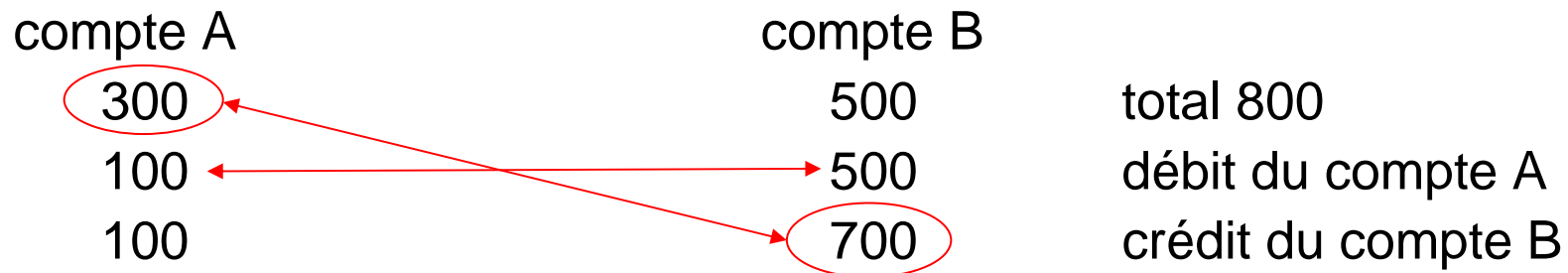
Une première difficulté est de définir ce qu'est un état global. Une première approche consiste à le définir comme étant un ensemble d'états locaux (les processus) à un instant donné. Cette définition suppose que l'on est capable de définir un même instant pour tous les processus (horloge globale).

Dans le modèle *happen-before*, un état global est un ensemble d'états locaux tous concurrents. Cette définition nécessite aucune hypothèse particulière sur le système pour pouvoir être utilisée (synchronisation, horloge globale).

Exemple

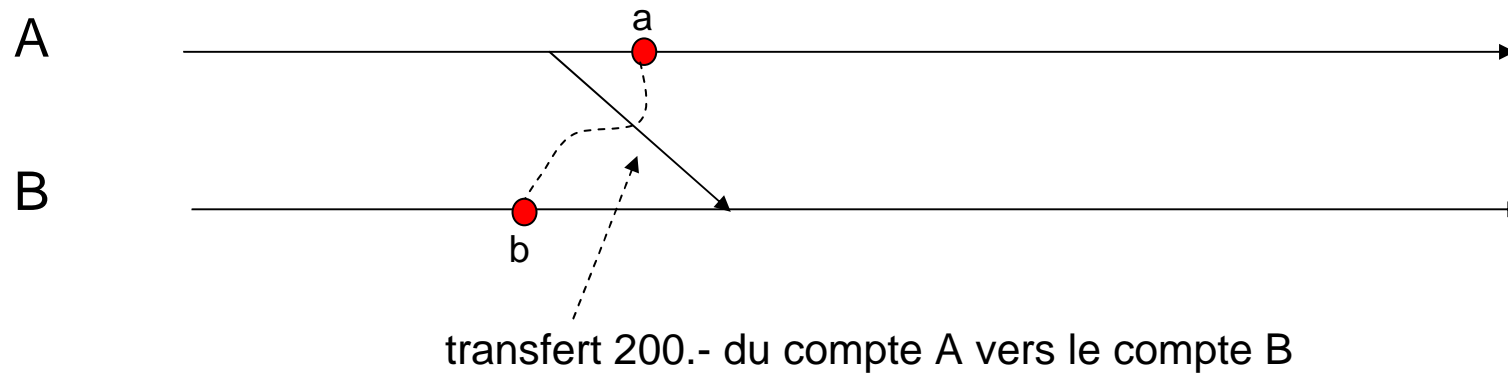
On considère un client qui possède 2 comptes en banques avec 300.- et 500.- sur chacun. En l'absence de mouvement, un instantané global doit nous permettre de conclure que le client possède au total 800.-.

Que ce passe-t-il si un transfert de 200.- est exécuté



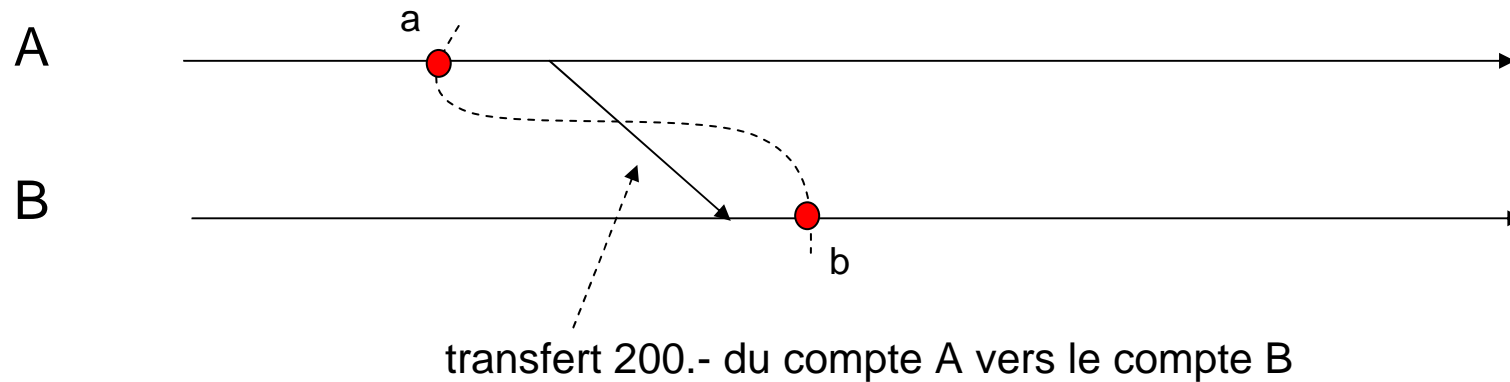
L'instantané global doit être cohérent avec la relation happen-before, de plus **les données transmises doivent être incluses dans l'instantané global.**

Exemple



Un exemple d'état global consistant. On connaît le solde du compte A (100.-), on mémorise qu'un message à été transmis correspondant au transfert de 200.- et on connaît le solde du compte B avant d'être crédité (500.-).

Exemple



La donnée des deux états a et b ainsi que le message correspondant au transfert n'est pas cohérent. Dans l'état a, le message n'est pas envoyé (le compte A pas nécessairement débité).

Définition

On suppose donné une exécution (E, \rightarrow) , c'est-à-dire un ensemble d'événements et une relation de précédence. Les événements d'un même processus sont ordonnés avec une relation totale $<$.

On définit une coupe comme étant un sous-ensemble $F \subset E$ satisfaisant

$$f \in F \wedge e < f \Rightarrow e \in F$$

Un coupe consistante ou un instantané global est un sous-ensemble $F \subset E$ satisfaisant

$$f \in F \wedge e \rightarrow f \Rightarrow e \in F$$

Algorithme de Chandy et Lamport

On suppose que les canaux de communications sont unidirectionnels et FIFO.

Les interfaces à implémenter sont:

```
public interface Camera extends MsgHandler {  
    void globalState(); // pour initier l'intsantané global  
}
```

```
public interface CamUser extends MsgHandler {  
    void localState(); // mémorise l'état d'un processus  
}
```

Algorithme de Chandy et Lamport

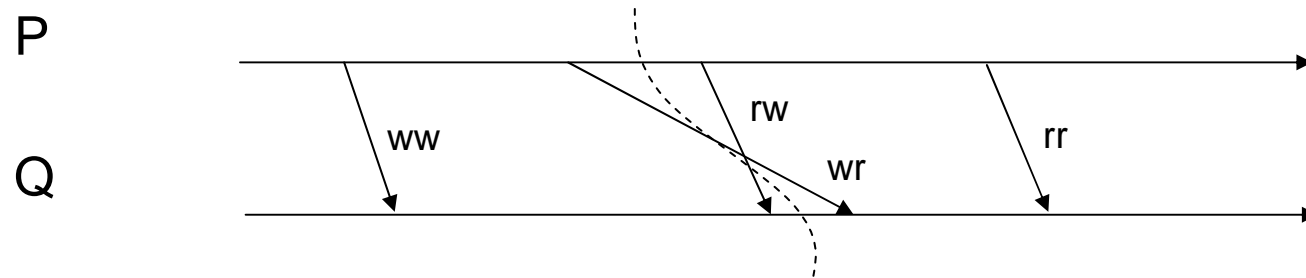
L'idée de l'algorithme est d'associer à chaque processus deux couleurs, blanc et rouge. L'état d'un processus associé à l'instantané global est celui dans lequel se trouve le processus juste avant de passer de blanc à rouge.

Un processus qui initie l'algorithme transmet des requêtes aux autres processus qui mémorisent leur état local et change leur couleur (blanc → rouge). Les changements de couleurs doivent vérifier

1. Les états locaux doivent être concurrents
2. L'état du canal doit être connu.

Pour cela, les processus utilisent un message dédié (*marker*), qui est transmis à tous les autres processus *avant* de transmettre les messages courants. Un processus qui reçoit un message *marker* devient rouge (ou le reste). On s'assure ainsi qu'un processus blanc ne reçoit jamais de message d'un processus rouge

Algorithme de Chandy et Lamport



messages ww: messages échangés avant que les processus soient impliqués

messages rr: messages échangés après l'instantané global

messages rw: ces messages correspondent à un instantané inconsistant, le *marker* prévient l'existence de tels messages

messages wr: ces messages composent l'état global du système.

Un processus P_j mémorise tous les messages reçus d'un processus P_i après être devenu rouge. Lorsque P_i devient rouge, il transmet un *marker* à P_j qui ne mémorise plus les messages.

Algorithme de Chandy et Lamport

```
public class RecvCamera extends Process implements Camera {  
    static final int white = 0, red = 1;  
    int myColor = white;  
    boolean closed[];  
    CamUser app;  
    LinkedList chan[] = null;  
    public RecvCamera(Linker initComm, CamUser app) {  
        super(initComm);  
        closed = new boolean[N]; // on arrête de mémoriser les messages oui/non  
        chan = new LinkedList[N]; // mémorise l'état du canal  
        for (int i = 0; i < N; i++)  
            if (isNeighbor(i)) {  
                closed[i] = false; // mémorisation  
                chan[i] = new LinkedList(); }  
            else closed[i] = true;  
        this.app = app;  
    }  
}
```

Algorithme de Chandy et Lamport

```
public synchronized void globalState() {  
    myColor = red;  
    app.localState();    // On mémorise l'état local  
    sendToNeighbors("marker", myId);    // on transmet le marker  
}
```

```
boolean isDone() {    // test si l'algorithme est terminé  
    if (myColor == white) return false;  
    for (int i = 0; i < N; i++)  
        if (!closed[i]) return false;  
    return true;  
}
```

Algorithme de Chandy et Lamport

```
public synchronized void handleMsg(Msg m, int src, String tag) {  
    if (tag.equals("marker")) {  
        if (myColor == white) globalState(); // changement de couleur  
        closed[src] = true; // on arrête de mémoriser les messages reçus  
        if (isDone()){ // l'algorithme est terminé  
            System.out.println("Channel State: Transit Messages ");  
            for (int i = 0; i < N; i++)  
                if (isNeighbor(i)) // affiche l'état des canaux  
                    while (!chan[i].isEmpty())  
                        System.out.println( ((Msg) chan[i].removeFirst()).toString());  
        }  
    } else { // message des applications, on mémorise  
        if ((myColor == red) && (!closed[src]));  
        chan[src].add(m); app.handleMsg(m, src, tag);  
        // on transmet le message à l'application concernée  
    }  
}
```

Canaux non FIFO

Si les canaux de transmission ne sont plus FIFO, le *marker* ne peut pas être utilisé tel quel pour indiquer la fin de la mémorisation des messages. De plus, on ne peut plus se contenter de transmettre une seule fois un *marker* car il peut arriver à un processus après des messages transmis plus tard.

La couleur d'un processus est incluse dans tous les messages, ainsi un processus peut déterminer si un message a été transmis avant ou après que le processus expéditeur ait changé de couleur.

Dans le *marker*, le processus expéditeur inclut le nombre de messages blanc préalablement transmis, permettant à un processus de déterminer

Impossibilité

Dans les réseaux asynchrones on ne peut pas résoudre le problème du consensus par un protocole (algorithme) qui tolère la panne d'un processus (au moins). Ce résultat peut être étendu considérablement.

On considère les *tâches de décision* qui déterminent une application de l'ensemble des états initiaux du système I dans un espace de décision D . A chaque état initiale, l'application associe un sous-ensemble de D qui correspond aux décisions valides.

Un tel protocole tolère une panne d'un processus si

1. le protocole s'exécute conformément à la spécification si tous les processus sont corrects
2. si un processus ne s'exécute pas correctement, les autres processus terminent leur exécution

Modèle

On considère un réseau dans lequel l'identité de tous les processus est connue et le graphe de communication est complet.

Les canaux de transmissions sont supposés fiables, les messages transmis arrivent avec un délai fini mais non borné (asynchrone).

définition: une tâche de décision T distribuée est une application

$$T : I^N \rightarrow 2^{D^N}$$

où N est le nombre de processus, I est l'ensemble des états initiaux des processus et D l'ensemble des états finaux.

Modèle

Un vecteur d'entrée x est un N-uple (x_1, x_2, \dots, x_N) où x_i est l'état initial du processus p_i .

Un vecteur de décision d est un N-uple (d_1, \dots, d_N) où d_i est l'état final du processus p_i .

On note D_T l'ensemble des vecteurs décision.

Exemple: Pour le problème du consensus on a

$$D_T = \{(0, 0, \dots, 0), (1, 1, \dots, 1)\}$$

Pour le problème de l'élection on a

$$D_T = \{(1, 0, \dots, 0), (0, 1, 0, \dots, 0), \dots, (0, \dots, 0, 1)\}$$

décisions adjacentes

définition: deux vecteur de décision d_1 et d_2 sont adjacents si ils sont différents pour une seule composante.

définition: le graphe de décision d'une tâche T à pour sommets les éléments de D_T et pour arêtes

$$E_T = \{(d_1, d_2) : d_1, d_2 \text{ sont adjacents} \}$$

définition: une tâche est non connexe si sont graphe de décision est non connexe.

Décisions partielles

Lorsque le protocole (algorithme distribué) s'exécute les processus corrects vont déterminer une valeur de décision. A un instant donné, il se peut qu'un sous-ensemble de processus n'ait pas encore pris de décision. Le vecteur (d_1, \dots, d_N) est un vecteur de *décision partiel*. On note les composantes de ce vecteurs pas encore définitive B-composantes.

Soit d_1 un vecteur de décisions, d_2 un vecteur de décision partiels est cohérent avec d_1 si ce dernier peut être obtenu en modifiant les B-composantes de d_2 .

définition: un protocole P résout une tâche T si pour chaque vecteur $d \in D_T$ il existe une exécution correspondante.

Résultat

Lemme: Soit T une tâche de décision et P un protocole qui résout la tâche et tolère un processus non correct.

Une exécution ou un processus est non correct s'arrête dans un état C et **les processus corrects déterminent un vecteur de décisions partiel qui est cohérent avec un vecteur de décision.**

Preuve: Supposons que le vecteur soit incohérent avec tous les vecteurs de décisions. Il se peut que le processus non correct soit seulement très lent... Après que $N-1$ processus aient pris leur décision et déterminé le vecteur de décisions partiel il commence à s'exécuter.

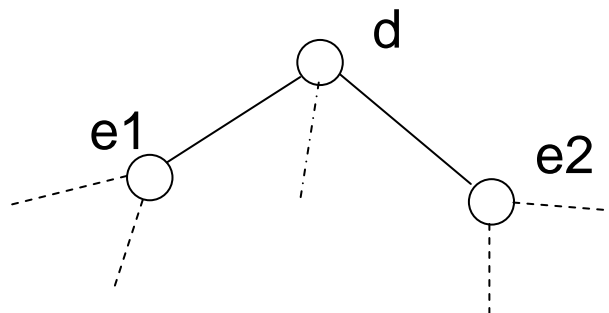
L'exécution finale détermine un vecteur de décision et tous les processus sont corrects.

Résultat

Lemme: Soit d un vecteur de décision, $N-1$ composantes de d déterminent la composante connexe de d .

preuve: On considère deux sous-ensemble de $N-1$ composante de d , ils déterminent deux vecteur de décisions partiels d_1 et d_2 .

Ces deux vecteurs sont cohérents avec les vecteurs de décisions e_1 et e_2 qui sont tous les deux adjacents à d .



Finalemment

Théorème: Une tâche de décision non connexe ne tolère pas de faute.

preuve: On note C_1, C_2, \dots, C_k les composantes connexes du graphe de décision. On montre que s'il existe un protocole P pour résoudre T tolérant alors il existe un protocole P' tolérant pour résoudre le problème du consensus.

Les processus exécutent le protocole P . Chaque processus qui prend une décision transmet la valeur à tous les processus. On suppose qu'un processus est non correct.

Au bout d'un temps fini, les $N-1$ processus corrects connaissent le vecteur de décisions partiel. Ils déterminent la composante connexe du vecteur de décision cohérent (les processus sont identifiés!). Tous les processus déterminent la même composante connexe et donc on a consensus sur la valeur choisie, **une contradiction.**