EPITA Alexandre DURET-LUTZ <adl@lrde.epita.fr>

ALDI — Algorithmique Distribuée

Notes de cours

Ce document est en cours de rédaction. Je le compléterai au fur et à mesure, en essayant de rester en avance par rapport au cours. Signalez-moi toute erreur par mail SVP.

1 Introduction **2**
1.1 Qu'est-ce qu'un système réparti 2
1.2 Exemples de grands systèmes répartis 3
1.3 Propriétés souhaitées 3
1.4 Les difficultés 4
1.5 Un aspect pratique : les middlewares 4
1.6 Les fondamentaux théoriques 4

2 Ordre et temps (dans un système réparti) **6**
2.1 Mesure du temps et relativité 6
2.2 Précision des horloges 8
2.3 Ordre et temps 9
2.4 Un modèle de système réparti (pour pouvoir travailler) 9
2.5 Relation de causalité 11
2.6 Horloges logiques 12
2.7 Horloges de Lamport 13
2.8 Ordre total 13
2.9 Application à l'exclusion mutuelle 13
2.10 Amélioration de Ricart et Agrawala 15
2.11 Mesure de complexité 16
2.12 Dépendance cachée 17
2.13 Horloges physiques 17
2.14 Le problème de la diffusion causale 19

Chapitre 1

Introduction

1.1 Qu'est-ce qu'un système réparti

Un *système réparti* est un ensemble d'éléments (processeurs, mémoires, capteurs, actionneurs...) autonomes qui *collaborent* pour la réalisation d'une tâche.

Par exemple une paire de talkie-walkies est un système distribué servant à transmettre un signal audio. Une équipe de foot est un système distribué qui cherche à marquer des points...

La collaboration entre les éléments du système implique l'existence d'un système de communication. Les talkie-walkies, utilisent des éméteurs/récepteurs radio, une équipe de foot utilise la parole et la vue (être visible à un endroit est une information qui est transmise visuellement au reste de l'équipe, qui prendra de décisions en conséquence). On pourrait aussi débattre des colonies de fourmis, du corps humain...

On s'intéressera plus particulièrement aux systèmes informatiques, pris à l'échelle des processus ou des machines. Un ensemble de serveurs SMTP qui se contactent pour se transmettre des messages est un système réparti. Un ensemble de processus qui communiquent par messages pour faire un calcul de même. Les voitures contiennent de plus en plus plusieurs calculateurs qui échangent des informations au travers d'un bus. [1]

Attention au vocabulaire employé :

Un système est dit parallèle lorsque les processus communiquent avec de la mémoire partagée (par exemple des threads).

Un système est réparti quand la mémoire n'est pas partagée : les communications se font par messages (selon les systèmes, les messages peuvent être envoyés de différentes façons, par exemple sous la forme de signaux inter-processus ou bien des paquets de données sur un réseau).

En anglais, *système réparti* se dit *distributed system*. À cause de l'anglais, on utilise parfois *système distribué* avec le même sens que *système réparti*. Certains auteurs préfèrent garder le terme *distribué* lié à la notion de calcul distribué (par exemple sur un *cluster*).

Leslie LAMPORT, que vous devez connaître comme le créateur de LaTeX, est avant tout un chercheur qui a énormément apporté au domaine des systèmes distribués. Il a donné la définition humoristique suivante d'un système distribué :

> « A distributed system is one that stops from getting any work done when a machine you've never heard of crashes. »

1. Par exemple un bus CAN : `http://fr.wikipedia.org/wiki/Controller_area_network`

Cette définition souligne une partie des problèmes qu'un système réparti doit résoudre au mieux : les problèmes d'interdépendance et de défaillances.

Une notion liée à l'existence d'un système de communication est celle de protocole de communication. Nous ne nous attarderons pas sur ce sujet (cf. cours de réseau) autrement que pour faire des hypothèses assez générales du genre « les canaux sont FIFO » (les messages envoyés d'une machine à une autre arrivent dans le même ordre si ils arrivent) ou « les canaux sont sans perte » (les messages envoyés arrivent à destination, et sans altération).

Le système de communication fait partie intégrante du système réparti. Par exemple quand on considère le système réparti qu'on appellera « le web », on considère non seulement tous les serveurs et les clients, mais aussi le réseau Internet qui connecte tout cela. Si l'on cherche à considérer l'*état global* du système à un instant T, il faut considérer non seulement l'état des différents serveurs et clients à cet instant, mais aussi l'état du réseau qui contient des messages en transit. Nous aurons l'occasion de revenir sur cette notion d'état global.

1.2 Exemples de grands systèmes répartis

Tous les services classiques de l'Internet sont de bons exemples : WWW, DNS, SMTP, ... Ils suivent une approche de type client/serveur, parfois avec des hiérarchies.

Des systèmes comme BitTorrent ont une architecture pair-à-pair où tout le monde peut être à la fois client et serveur, et il n'y a pas de véritable maître.

Les systèmes embarqués sont de plus souvent répartis. On l'a déjà dit à propos des voitures, mais c'est encore plus vrai avec des transports comme les avions ou les trains dont les systèmes embarqués mobiles doivent aussi communiquer avec des systèmes embarqués fixés au sol.

Les réseaux de capteurs sans fils [2] sont comme leur nom l'indique des réseaux *ad hoc* dont les nœuds sont des capteurs qui récoltent et transmettent les données récoltées. Un tel réseau doit savoir s'organiser tout afin de pouvoir être "semé" de façon imprécise.

Le réseau constitué des satellites GPS (Global Positioning System) et des récepteurs GPS est un autre exemple de système réparti. Ici les clients reçoivent des informations de plusieurs serveurs (les satellites) pour déterminer leur position.

1.3 Propriétés souhaitées

Que peut-on souhaiter d'un système distribué ?

- Survivre aux défaillances (par exemple en entroduisant de la redondance dans les communications pour pouvoir corriger les erreurs de transmissions, ou de la redondance de machines pour pouvoir remplacer une machine défaillante), peut-être à travers un mode dégradé du système. Ceci suppose d'être en mesure de détecter les défaillances.
- Résister aux perturbations des communications, qui peuvent être de plusieurs natures : perte ou altération de messages, déconnexions, changement du débit ou de la latence...
- Résister aux attaques. On pense ici à la sécurité du système : problèmes d'intégrité, de confidentialité, résistance aux de service...
- Passer à l'échelle (= *scalability*) c'est-à-dire préserver les performances malgré la croissance du nombre d'éléments du système, ou de l'étendue géographique du système.
- S'adapter au changement (d'environnement, d'utilisateur). Ce point traduit qu'un bon système distribué, doit pouvoir être modifié, étendu assez facilement.

Notez que certains de ces souhaits ne sont pas spécifiques aux systèmes distribués.

2. http://fr.wikipedia.org/wiki/Réseau_de_capteurs_sans_fil

1.4 Les difficultés

Elles sont liées aux point précédents.

L'utilisation de communications asynchrones (un message est envoyé au travers le réseau, mais on ne sait pas quand il arrivera) fait qu'on n'a pas de borne supérieure sur les délais de transmission. On parle d'*unbounded determinism*. Sans cette borne, il est donc difficile de détecter les défaillances.

Le caractère dynamique du système (pensez à un réseau de capteurs sans fils installé sur un troupeau de moutons et tout ce qui peut arriver : des petits groupes qui se séparent du troupeau et reviennent ensuite, des capteurs qui se font grignoter...) fait qu'il est difficile de l'administrer, de le debugger, de capturer un *état global*.

Les grands systèmes posent des problèmes de passage à l'échelle aussi bien d'un point de vue physique (les canaux de communication sont ils assez larges/rapides ?) que d'un point de vue logique (les protocoles de communication et les algorithmes utilisés sont ils capables de gérér un grand nombre de nœuds ?).

1.5 Un aspect pratique : les middlewares

Un *middleware* (ou *intergiciel* pour ceux qui préfèrent inventer des mots pour ne pas emprunter aux langues étrangères) est un couche logicielle qui se place entre l'application et le système d'exploitation, pour offrir des services utiles à la construction de systèmes répartis. On peut les voir comme des super-systèmes d'exploitation avec des primitives de communication de plus haut niveau.

Les middlewares de type RPC (*Remote Procedure Call*) permettent des appels de procédures/fonction distantes, le plus souvent de façon synchrone. RPC de Sun permet de faire de tels appels en C et est utilisé par exemple par NFS. Utilisez JavaRMI en Java. CORBA est un standard de l'OMG (Object Management Group) pour les appels distants avec une approche orientée objet.

Il existe aussi des middlewares dit *message oriented*, pour des communications asynchrones utilisant des files de messages, ainsi que du broadcast/multicast. Par exemple Java Message Services.

Ces middlewares offrent aussi des services d'annuaire, permettant de trouver un service dans le système réparti.

Nous nous intéresserons à des problèmes plus théoriques.

1.6 Les fondamentaux théoriques

Le sujet de ce cours : expliquer les principes théoriques à la base des systèmes répartis, les problèmes, les solutions et limites connues.

Par exemple :
- qu'est-ce qu'une exécution répartie ?
- qu'est-ce qu'un état global ?
- comment déterminer des propriété globales à partir d'observations locales ?
- comment des processus/machines peuvent-elles partager des données/informations sans disposer de mémoire commune ?
- comment maintenir la cohérence de ces données réparties ?
- existe-t-il des problèmes insolubles ?
- comment supporter des défaillances partielles ?
- ...

Pour cela on travaille à partir de modèles (des abstractions) d'applications réparties. Ce n'est pas la réalité, mais on espère s'en approcher.

Chapitre 2

Ordre et temps (dans un système réparti)

La notion de temps dans un système réparti est assez complexe parce que chaque composant du système peut percevoir le temps de façon différente.

Imaginons un système de réservation de billet d'avion. Que se passe-t-il si deux agences de voyage réservent la dernière place d'un vol ? Idéalement la première des agence a avoir émis l'ordre de réservation devrait obtenir satisfaction, mais les délais de transmission sur le réseau peuvent très bien faire en sorte que la seconde réservation arrive avant la première.

Si chaque agence possède sa propre horloge, et l'utilise pour dater les demandes, rien ne prouve que ces horloges sont bien synchronisées.

Après une brève digression sur la relativité (pour se convaincre qu'il ne sert à rien d'espérer des horloges synchrones) nous étudierons plusieurs mécanismes qui permettent de définir un ordre sur les événements d'un systèmes réparti, sans pour autant faire intervenir le temps physique.

2.1 Mesure du temps et relativité

On apprend au lycée que les vitesses s'additionnent $u = v + w$: si un homme avance dans un train à la vitesse $v = 5km/h$, et que ce train avance lui-même à la vitesse $w = 150km/h$, l'homme se déplace à $u = v + w = 155km/h$ par rapport à la surface de la Terre. Un calcul similaire peut être fait pour une balle de fusil tirée depuis une voiture en déplacement.

Ces calculs sont incorrects dès que les vitesses manipulées approchent de celle de la lumière dans le vide, supposée constante dans la théorie relativiste.

La vitesse de la lumière est $c = 299\,792\,458m/s$ dans le vide. [1] Si cette lumière est projetée à partir d'un objet en mouvement, par exemple le phare avant d'un train, cette vitesse est toujours la même, c, car elle ne peut pas être dépassée. [2] Le principe d'addition des vitesses $u = v + w$ (celle du train et de la lumière) ne marche plus ici. La formule donnée par la relativité restreint est

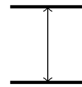
$$u = \frac{v + w}{1 + \frac{vw}{c^2}}$$

1. Dans les autres milieux, il faut diviser par le coefficient de réfraction : $1,0003$ pour l'air, $1,33$ pour l'eau salée.
2. Plus formellement, la vitesse de la lumière dans le vide est c quel que soit le référentiel choisi (p.ex. dans le référentiel du train en mouvement ou dans celui de la terre)

(Vérifiez que si v ou w est égal à c, alors $u = c$.)

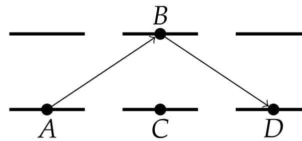
Le rapport avec le schmilblick ? Il est double. D'une part la vitesse de la lumière nous donne une borne sur la vitesse de transmission d'une information (=latence) dans n'importe quel réseau, et donc dans n'importe quel système réparti.[3] D'autre part la relativité restreinte nous dit aussi que le temps s'écoule différemment lorsqu'on se déplace : un système réparti aura donc du mal a garder ses horloges synchrones.

Il existe un formule qui donne la dilatation du temps en fonction de la vitesse de déplacement, mais je trouve plus simple de retenir une façon de retrouver cette formule. L'idée est d'utiliser une montre à photons imaginaire pour mesurer le temps.



Un photon est envoyé de la plaque du bas vers la plaque réfléchissante du haut. Cette dernière renvoie le photon sur la plaque du bas. On mesure alors le temps d'aller-retour pour obtenir la valeur $2 \cdot t_{fixe}$. Cette valeur est constante, et les "tops" du photon permet de mesurer le temps. Le photon se déplaçant à la vitesse c on peut aussi déduire de cette mesure l'écartement des deux plaques : $c \cdot t_{fixe}$.

Imaginons maintenant que la montre se déplace vers la droite à la vitesse v. Un aller-retour du photon ressemble alors à ceci :



Notons $2 \cdot t_{mobile}$ le temps d'aller-retour mesuré entre A et D.
La distance AB parcourue par le photon est $c \cdot t_{mobile}$.
La distance AC parcourue par la montre est $v \cdot t_{mobile}$.
La distance BC entre les deux plaques est toujours $c \cdot t_{fixe}$.

En appliquant le théorème de Pythagore :

$$\begin{aligned}(AB)^2 &= (AC)^2 + (BC)^2 \\ (c \cdot t_{mobile})^2 &= (v \cdot t_{mobile})^2 + (c \cdot t_{fixe})^2 \\ t_{mobile}^2 &= \left(\frac{v}{c} t_{mobile}\right)^2 + t_{fixe}^2 \\ t_{mobile} &= \frac{t_{fixe}}{\sqrt{1 - \left(\frac{v}{c}\right)^2}}\end{aligned}$$

Le temps t_{mobile} mesuré par la montre en mouvement du point de vue du référentiel fixe est donc $\gamma = \frac{1}{\sqrt{1-\left(\frac{v}{c}\right)^2}}$ fois plus grand que le temps t_{fixe} mesuré dans le référentiel de la montre (qui est alors immobile). Ce facteur γ est le *facteur de Lorentz*.

3. Par exemple le champ électrique, qui conduit l'information le long d'un fil de cuivre, se déplace à la vitesse de la lumière dans le vide divisée par le coefficient de réfraction du cuivre : 1, 1. C'est très rapide, mais il faut prendre compte du fait que l'atténuation du signal est très forte, et qu'un champ magnétique est très sensible aux perturbations. Le temps est réellement perdu dans les équipements intermédiaires qui servent à régénérer le signal, et à gérer les protocoles.

Exemple. Calculons γ pour un voyage en avion à $720km/h$. On a $v = 2.10^2$, $c = 3.10^8$ (environ), donc $\frac{v}{c} = \frac{2}{3}10^{-6}$ et $\left(\frac{v}{c}\right)^2 = \frac{4}{9}10^{-12}$. Si x voisin de 0, on a $\frac{1}{\sqrt{1-x}} = 1 + \frac{x}{2} + o(x^2)$. Donc $\gamma \approx 1 + \frac{1}{2}\left(\frac{v}{c}\right)^2 = 1 + \frac{2}{9}10^{-12}$. La dilatation est donc très faible. Si l'on prend $t_{fixe} = 10h$ on trouve $t_{mobile} = \gamma t_{fixe} = 10h + 8ns$. t_{fixe} correspond au temps écoulé pour une personne dans l'avion (dans sont référentiel la montre est fixe), alors que t_{mobile} correspond au temps sur terre (par rapport à laquelle la montre est mobile). On vieillit donc moins vite à bord d'un avion.

Les satellite GPS, qui se déplacent à $14000km/h$ autour de la Terre, sont beaucoup plus sensible à cette dilatation du temps puisque leur vitesse est plus grande : les horloges embarquées subissent un retard de $7,2\mu s$ par jour. Cependant ce n'est pas leur seul problème. À ce principe de relativité restreinte, il faut ajouter un autre principe de la relativité générale : l'écoulement du temps est aussi affecté par la gravité. À une altitude de $20000km$ la gravité est moins forte est le temps passe cette fois-ci plus vite ! Les horloges atomiques embarquées dans ces satellites prennent $45\mu s$ d'avance par jour. En combinant ces deux effets, c'est donc $38\mu s$ que les satellites GPS doivent retrancher chaque jour. Sans cette correction, l'erreur des positions GPS augmenterait de $12km$ par jour.

Ces problème d'horloges existent aussi entre les horloges disposées à différentes altitudes terrestres (que ce soit en haut d'une montagne, ou même à l'échelle d'un immeuble), même si la dilatation est beaucoup plus faible.

2.2 Précision des horloges

Ce problème de la dilatation du temps reste assez théorique pour des systèmes répartis à petite échelle. Un autre problème est celui de la *précision* des horloges.

Pour construire une horloge, il faut déjà savoir quoi mesurer, puis réussir à le faire précisément.

La seconde a d'abord été définie comme $1/86400$ du jour solaire terrestre moyen (la durée des jours n'étant pas exactement constante). Pour tenir compte des imperfections de la rotation de la Terre qui ralentit notamment à cause des marées, elle a été définie en 1956 comme $1/31556925,9747$ de l'année 1900.

Depuis 1967, la seconde est la durée de 9 192 631 770 périodes de la radiation correspondant à la transition entre les niveaux hyperfins $F = 3$ et $F = 4$ de l'état fondamental $6S_{1/2}$ de l'atome de césium 133 au repos à 0 degré Kelvin. C'est l'unité du S.I. la plus précisément connue avec une précision de 10^{-14}.

La précision des horloges s'est aussi améliorée avec le temps. En 1671, Huygens utilise un pendule pour battre les secondes [4] et un système de roues dentées pour compter les secondes. La *dérive* du pendule de Huygens est inférieure à $10s$ par jour.

Les montres mécaniques fonctionnent de façon similaire avec une fréquence de battement de l'ordre de 3Hz ou 4Hz (donc une meilleure précision, mais pas forcément une meilleure dérive).

Les horloges de Shortt (double balancier sous vide) furent inventées dans les années 1920–1930 pour les besoins des laboratoires astronomiques. Elle possèdent une dérive inférieure à $2ms$ par jour.

Vinrent ensuite les horloges à Quartz. Un cristal de Quartz soumis à une charge électrique oscille. En fonction de la découpe du cristal, on peut choisir la fréquence des oscillations, par exemple $32768Hz$. Il reste à accoler un petit circuit électrique pour compter les oscillations. On obtient une précision inférieure à la micro seconde, et une dérive inférieure à $1s$ par jour. Les horloges à

4. À paris, un pendule de 99,4cm bat la seconde. Il faut un pendule de 99,1cm à l'équateur ou 99,5cm aux pôles pour obtenir le même rythme.

quartz sont calibrées pour une température donnée (par exemple 25 degrés Celsius) et la dérive augmente de façon quadratique lorsqu'on s'écarte de cette température.

Le fin du fin est l'horloge atomique, qui atteint une précision de 10^{-14}. En France, le Bureau international des poids et mesures, à Sèvres, définit le Temps Atomique International en faisant la moyenne d'horloges atomiques réparties sur la planète. Les satellites GPS embarquent aussi chacun une horloge atomique.

2.3 Ordre et temps

Lorsqu'on considère les événements d'un système réparti, on se pose souvent des questions d'ordre : « est-ce que tel événement précède tel autre événement ? ».

Exemple concret : deux sections critiques SC_1 et SC_2 sont en exclusion mutuelle si $Fin(SC_1)$ précède $Fin(SC_2)$ ou bien si $Fin(SC_2)$ précède $Fin(SC_1)$.

L'existence d'un temps universel (connu de tous les éléments du système) permettrai de dater les événements et de répondre à cette question, mais la difficulté d'avoir des horloges précises et synchrones doit nous faire réfléchir à une notion d'ordre qui ne dépende pas d'un temps universel.

Pour coordonner les activités du système, on souhaite définir un ordre sur les événements. Cette notion d'ordre doit être implémentée en tenant compte des difficultés suivantes :
– pas de mémoire commune aux processus du systèmes,
– pas d'horloge commune (éventuellement chaque composant possède sa propre horloge),
– des communications asynchrones (pas de borne sur les délais de transmission),
– des traitements asynchrones (chaque processus avance à sa propre vitesse, sans borne sur le rapport des vitesses entre les processus).

2.4 Un modèle de système réparti (pour pouvoir travailler)

On note E, F, ... les processus du système réparti (processus n'est pas a prendre au sens Unix ici, il s'agit juste d'un élément du système, cela pourrait être une machine entière contenant plusieurs applications qui réalisent le processus).

Ces processus communiquent exclusivement en s'envoyant des messages.

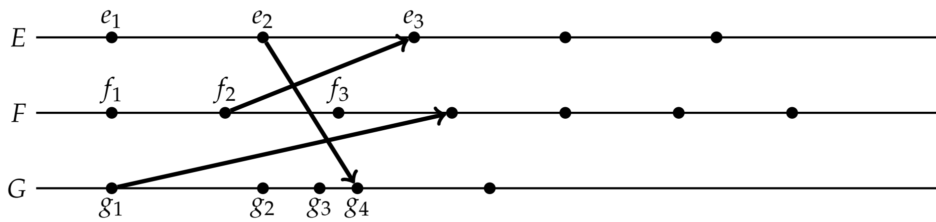
Dans la vie d'un processus peuvent se produire trois types d'événements :
– des événement locaux (le processus change d'état)
– des émissions de messages
– des réceptions de messages

Ces événements, quelque soit leur type, seront notés $e_1, e_2, \ldots$ pour le processus E ; $f_1, f_2, \ldots$ pour le processus F, etc.

Comme les messages ne peuvent transiter plus vite que la vitesse de la lumière, on suppose que la réception d'un message ne peut pas être faite en même temps que son émission : il y a forcément un délai, quand bien même il serait infime. D'autre part les communications étant asynchrones, nous n'avons pas de borne supérieure sur le temps de transit d'un message. Pour l'instant nous supposerons que le réseau de communication est fiable : les message arrivent toujours, et ils arrivent intacts.

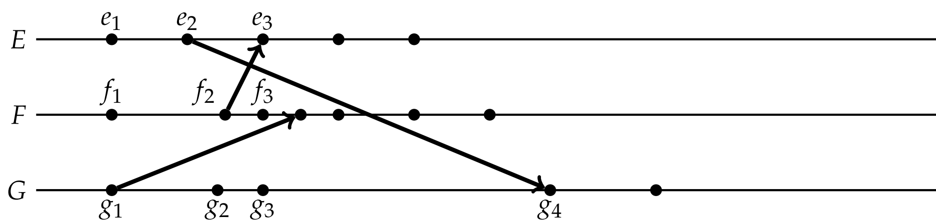
Enfin, chaque processus avance à sa propre vitesse, qui n'est pas forcément constante.

Un diagramme des événements d'un système de trois processus pourrait prendre la forme suivante :

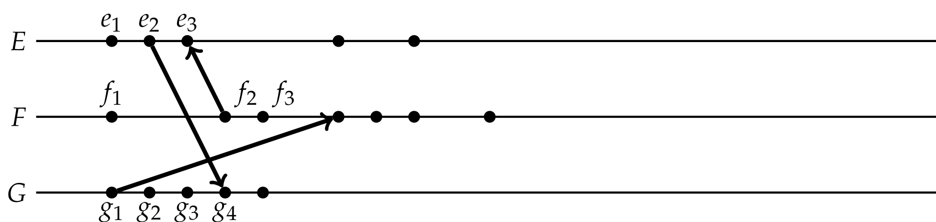


Le temps s'écoule ici vers la droite [5]. Les points noirs correspondent à des événements des processus. Les flèches entres des événements de deux processus différents correspondent à un envoi de message : l'événement source de la flèche est bien sûr l'expédition tandis que l'événement pointé par la flèche est la réception du message. Les événements qui ne sont pas connectés à une flèche sont les événements locaux des processus.

Sur ce diagramme, le rapport entre l'écoulement du temps entre les différents processus est purement arbitraire. Sur le schéma, f_2 paraît un peu plus à gauche que e_2, mais ce n'est pas pour cela que f_2 précède e_2 en pratique. Selon la vitesse d'exécution de F et E, on pourrait très bien avoir une exécution qui ressemble plus à



En fait, du point de vue d'un concépteur d'algorithme distribué, qui n'a aucune idée des vitesse de chaque processus, ces deux schémas sont équivalents. Chaque axe du temps devant être interprété isolément il ne serait pas non plus choquant de dessiner le schéma suivant, où le message de f_2 à e_3 **semble** revenir dans le passé.



En pratique, on préférera tout de même dessiner des figures telles que les flèches sont dirigées vers la droite.

L'*histoire d'un processus*, appelée aussi la *trace d'un processus* est la suite d'événements ordonnés par l'horloge locale du processus.

$$\text{trace}(E) = [e_1, e_2, e_3 \ldots]$$

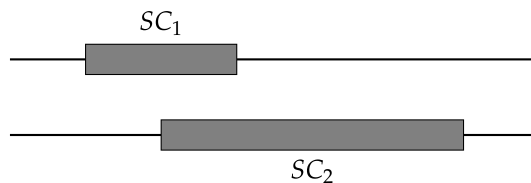
Les trois derniers diagrammes d'événements sont équivalents car chaque processus y a la même trace.

5. Certains auteurs présentent ces diagrammes verticalement, avec le temps s'écoulant vers le bas.

Au sein de l'histoire d'un processus, tous les événements sont ordonnés. Entre deux processus, c'est une autre histoire...

Synchroniser deux processus revient à imposer un ordre entre des événements appartenant aux deux histoires.

Par exemple le problème de l'exclusion mutuelle : deux processus ne doivent pas entrer dans leur section critique en même temps. Il faut donc imposer que $Fin(SC_1)$ précède $Fin(SC_2)$ ou l'inverse que $Fin(SC_2)$ précède $Fin(SC_1)$.



Cette contrainte doit de plus être respectée par les processus qui n'ont qu'une vision locale du système (et du temps).

Deux problèmes :

1. être capable de définir une relation globale de précédence entre les événements (« ça c'est produit avant »)
2. pouvoir ordonner deux événements à partir d'information locales (un processus n'a pas la vision globale du système)

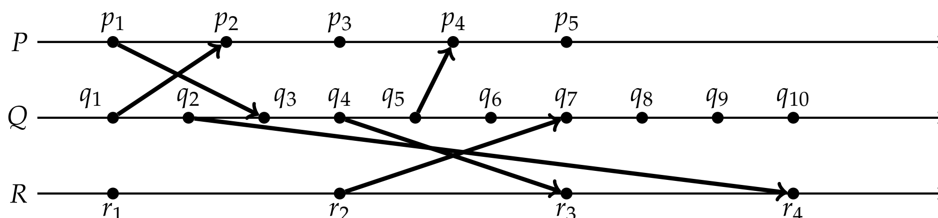
Une solution à ces problèmes fut proposée par Leslie LAMPORT dans un article facile à lire : « Time, Clocks, and the ordering of Events in a Distributed System ». Les prochaines sections s'appuient sur cet article.

2.5 Relation de causalité

On sait ordonner partiellement certains événements. Notons $a \rightarrow b$ la relation de causalité entre deux événements. On peut dire que

- $a \rightarrow b$ si a et b sont deux événements d'un même processus, et que a se produit avant b.
- $a \rightarrow b$ si a est une émission de message et b est une réception.
- $a \rightarrow b$ si il existe c tel que $a \rightarrow c$ et $c \rightarrow b$.

Sur un exemple :



(Note : sur ce diagramme le canal de communication entre Q et R n'est visiblement pas FIFO.)

On a ici $q_1 \rightarrow q_2 \rightarrow q_3 \rightarrow \cdots$
Mais aussi $p_1 \rightarrow q_3$ et $q_4 \rightarrow r_3$ donc $p_1 \rightarrow r_4$.
Par contre $r_1 \not\rightarrow p_4$.

On note $e_1 \| e_2$ (qu'on lit « e_1 *concurrent avec* e_2 ») lorsque $e_1 \not\rightarrow e_2$ et $e_2 \not\rightarrow e_1$.

On a par exemple $p_3 \| q_3$. Le processus P ne peut pas savoir ce que Q a fait en q_3 tant qu'il n'a pas reçu de message en p_4.

2.6 Horloges logiques

De façon abstraite, une horloge permet d'horodater les événements, mais cette date est un simple numéro qu'on ne suppose pas lié au temps physique (d'où le nom d'horloge logique).

Chaque processus P_i possède une horloge C_i qui numérote les événements. On notera $C_i(a)$ la date de tout événement a de P_i. (C_i n'est bien sûr pas capable de dater les événements des autres processus.)

L'ensemble de ces horloges (appelé système d'horloge) permet de définir une horloge globale C telle que $C(a) = C_i(a)$ si a est un événement de P_i. Cette horloge globale est purement abstraite car les processus n'ont accès qu'à leur propre C_i.

Nous allons par la suite définir plusieurs systèmes d'horloges, c'est-à-dire des implémentations de C_i pour chaque processus. Pour être correct, ces systèmes doivent vérifier la condition suivante.

Définition 1 (Condition d'horloge)**.** *Un système d'horloges C respecte la* condition d'horloge *si pour tous événements a et b on a*

$$a \rightarrow b \implies C(a) < C(b)$$

Autrement dit si b dépend causalement de a, sa date donnée par C est ultérieure à celle de a.

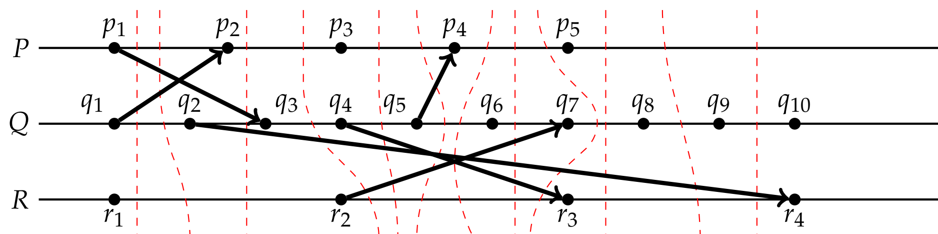
Note : On ne peux pas supposer l'équivalence car alors on aurait $a \| b \iff C(a) = C(b)$. Dans le diagramme d'événements précédent, on avait $p_2 \| q_3$ et $p_3 \| q_3$ donc les trois événements devraient avoir la même date ; mais aussi $p_2 \rightarrow p_3$ donc ces deux événements devraient avoir des dates différentes.

Par définition de $\rightarrow$, il est évident que la condition d'horloge est vérifiée si les deux conditions suivantes sont remplies :

(C1) Pour a, b deux événements du processus P_i, si a précède b alors $C_i(a) < C_i(b)$.

(C2) Si a est une émission d'un message par P_i et que b est la réception correspondante par P_j, alors $C_i(a) < C_j(b)$.

Si l'on suppose que l'on dispose d'une telle horloge C, on peut représenter ses tics :



La condition **(C1)** signifie qu'il doit y avoir au moins 1 tic entre chaque événement d'un même processus. La condition **(C2)** signifie que chaque message doit traverser un tic.

Cette figure n'est qu'un possibilité parmi beaucoup d'horloge C possibles. Les tics ne se traduisent pas nécessairement par des traits verticaux puisque les horloges de chaque processus peuvent avancer à des vitesses différentes. En changeant les échelles de temps de chaque processus comme

on a déjà fait, on pourrait obtenir une autre représentation de ce modèle où les tics sont bien des traits verticaux.

Nous avons donc deux conditions suffisantes **(C1)** et **(C2)** qu'une implémentation doit satisfaire pour respecter la condition d'horloge.

Cherchons maintenant à implémenter une telle horloge.

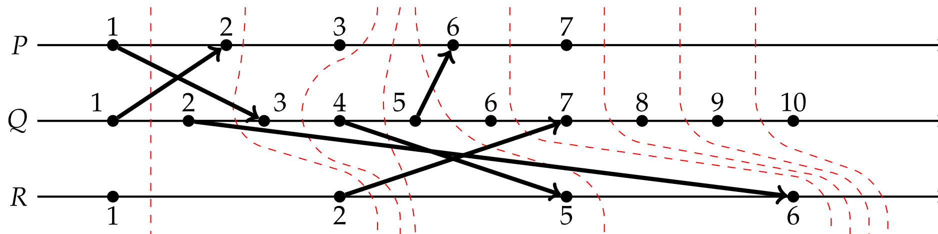
2.7 Horloges de Lamport

On suppose que les processus sont des algorithmes qu'on va équiper d'horloge. Chaque processus P_i possède une variable C_i qui représente l'horloge. La valeur de C_i va changer entre les événements de l'algorithme, mais ce changement n'est pas lui-même considéré comme un événement. Deux règles d'implémentation permettent de garantir **(C1)** et **(C2)** :

(IR1) P_i incrémente C_i entre deux événements.

(IR2) (a) Lorsque P_i envoie un message m, ce qui correspond à un événement a, le message contient (en plus du reste) la valeur timestamp $= C_i$.
(b) Quand P_j reçoit le message m, il fait $C_j \leftarrow \max(C_j, \text{timestamp} + 1)$ et considère que le message a été reçu à cette nouvelle date.

Ci-dessus, les noms des événements ont été remplacés par les valeurs des horloges locales, initialisées à 1 pour le premier événement, et mise à jour avec les règles ci-dessus.



2.8 Ordre total

Dans ce dernier exemple, l'ordre induit par les horloge est partiel. Par exemple p_3 et q_3 ont la même date. Comme deux événements de dates identiques appartiennent forcément à des processus différents, on peut imposer un ordre total, noté $\Rightarrow$, en numérotant les processus de façon arbitraire.

$$a \Rightarrow b \iff \begin{cases} \text{soit } C(a) < C(b) \\ \text{soit } C(a) = C(b) \text{ et } P_i \prec P_j \end{cases}$$

Où $\prec$ est un ordre sur les processus.

Cet ordre total nous sera parfois utile.

Notez que $\Rightarrow$ dépend de C et de $\prec$ donc est loin d'être unique.

2.9 Application à l'exclusion mutuelle

Supposons qu'un ensemble de processus partagent une unique ressource. On veut un algorithme de synchronisation qui fait en sorte que

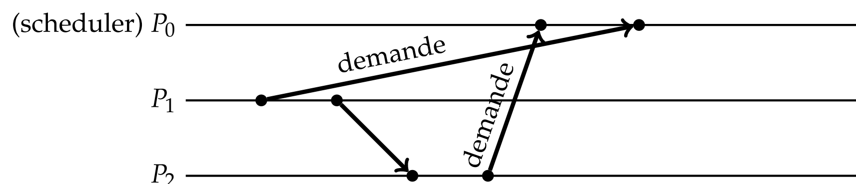
(1) Un processus qui a obtenu l'autorisation d'accès à la ressource la libère avant qu'un autre processus puisse obtenir cette autorisation.

(2) Les différentes demande d'accès sont servies dans l'ordre où elles ont été émises (ce qui n'est pas forcément l'ordre dans lequel elles ont été reçues).

(3) Si tout processus qui a eu la ressource finit par la libérer, alors tout processus qui demande la ressource finira par l'obtenir.

Au début, on suppose que l'accès à la ressource est donné à un processus.

Un tel algorithme n'est pas simple à réaliser. Par exemple supposons l'existence d'un processus P_0 qui fasse office d'allocateur de ressource (c'est lui qui reçoit les demandes, et délivre les autorisations), et considérons le scénario de demandes suivant : P_1 envoie une demande d'accès à P_0 puis un message quelconque à P_2 ; suite à ce message P_2 envoie aussi une demande d'accès à P_0. Les délais de transmission n'étant pas connus, on peut supposer que $_0$ reçoit le message de P_2 avant celui de P_1.



Comment doit se comporter P_0 ? À la réception de la demande de P_2 il ne sait pas qu'une demande antérieure arrive de la part de P_1. S'il décide de donner la ressource à P_2, il ne respecte pas la propriété (2), qui demande que les requêtes soient traitées dans l'ordre de leurs émissions.

Nous allons résoudre se problème en utilisant un système d'horloge locales (règles **(IR1)** et **(IR2)**) pour définir un ordre total sur les événements (donc en particulier sur les demandes), puis en faisant en sorte que chaque processus soit mis au courant des opérations des autres. L'algorithme ainsi construit ne nécessite pas de processus particulier pour gérer l'allocation de la ressource.

Pour simplifier la description de l'algorithme, on suppose que les canaux sont FIFO et sans perte. Ces supposions peuvent-être levées en introduisant des numéros de messages et un protocol d'accusé de reception (comme dans TCP). On suppose aussi que tous les processus peuvent s'envoyer des messages.

Chaque processus possède sa propre file de requête, une structure de donnée privée pour garder trace des requêtes d'accès à la ressource.. Chaque file contient initialement le message "T_0 : P_0 requests resource" où P_0 est l'identifiant du processus qui a la ressource initialement, et T_0 est une date plus petite que les valeurs initiales de toutes les horloges.

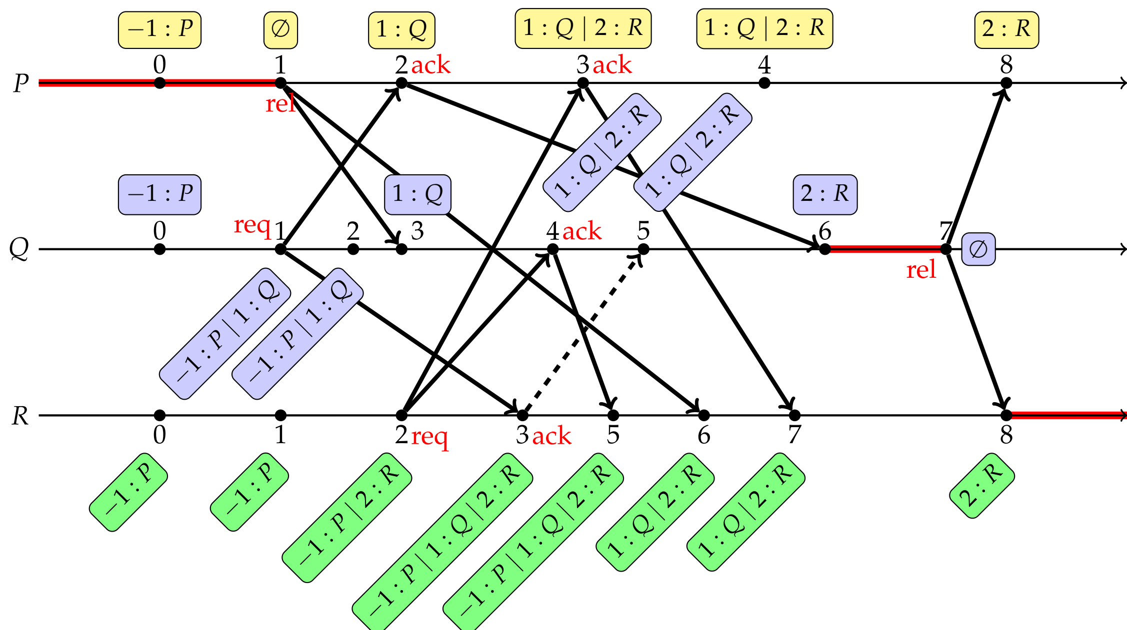
L'algorithme est composé de 5 règles, et les actions de chaque règle sont atomiques (c'est-à-dire vues comme un seul événement).

1. Pour demander une ressource, P_i envoie le message "C_i : P_i req" à tous les processus et ajoute ce message à sa propre file.
2. Quand P_j reçoit un message "T_m : P_i req", il l'ajoute à sa file et envoie une réponse "C_i : P_j ack" à P_i.
   (Cette réponse est facultative si P_j a déjà envoyé un message à une date ultérieure à T_m.)
3. Pour libérer une ressource, P_i retire le message "T_m : P_i req" de sa file et envoie le message "C_i : P_i rel" à tous les autres processus.
4. Quand un processus P_j reçoit "T_m : P_i rel", il retire la requête correspondante de sa file. Aucune réponse n'est envoyée.
5. P_i considère qu'il possède la ressource si et seulement si les deux conditions suivantes sont réunies :

(1) il possède un message "T_m : P_i req" dans sa file, et ce message précède tous les autres au sens de l'ordre total $\Rightarrow$,

(2) P_i a reçu un message de tous les autres processus avec in timestamp strictement supérieur à T_m.

Notez que ces deux conditions constituent un test local à P_i.



Attention, cette figure ne représente pas toutes les données : afin de pouvoir décider s'il a accès à la ressource, chaque processus doit de plus conserver un tableau des dates d'expédition des derniers messages reçus de chaque processus.

Un processus A qui a reçu des "ack" de tout le monde et qui est en tête de sa liste de demande a-t-il toujours raison d'accéder à la ressource ? Pour que ce soit incorrect, il faudrait qu'un autre processus B ait demandé l'accès à la ressource avant A. Si B a demandé l'accès, il a forcément envoyé une requête à tout le monde, et comme les canaux sont FIFO, A a du recevoir la requête de B avant de recevoir l'"ack" de B à sa propre requête.

En pratique l'"ack" ne sert à rien d'autre qu'à garantir qu'il n'y a pas de requête antérieure en transit sur le canal. Ce message "ack" peut-être omis si le processus qui doit l'envoyer à déjà écrit au destinataire avec une date ultérieur à la requête. Sur l'exemple ci-dessus le message en pointillés est superflu car la requête de Q, envoyée juste avant, possède déjà une date ultérieure à la requête de P.

2.10 Amélioration de Ricart et Agrawala

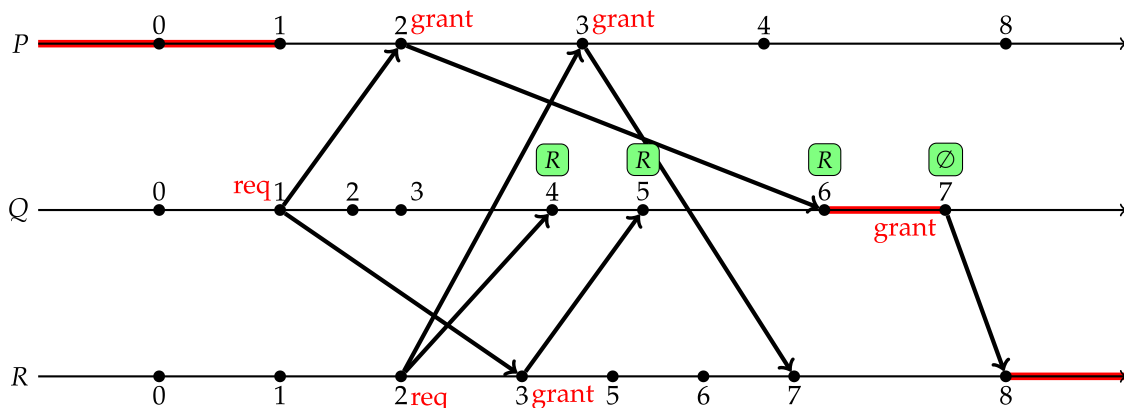
Avec l'algorithme précédent, lorsqu'un processus en section critique reçoit une demande il va d'abord répondre "ack" au processus, puis plus tard il quittera sa section critique, et enverra "release" à tout le monde. On remarque que l'envoi du "ack" au processus qui réclamait l'accès est inutile.

L'idée de Ricart et Agrawala est de fusionner "release" et "ack" en un seul type de message : "grant".

Chaque processus possède une file d'attente de demandes non encore autorisées, et les règles deviennent les suivantes :

1. Pour demander une ressource, P_i envoie le message "C_i : P_i req" à tous les autres processus
2. Quand P_j reçoit un message "T_m : P_i req", deux situations sont possibles :
   (a) Si P_j n'a pas demandé la ressource, il répond à P_i avec le message "C_j : P_j grant"
   (b) Si P_j a lui-même demandé la ressource à une date T_j, à nouveau deux sous-cas sont possibles :
      i. Si $(T_j, j) < (T_m, i)$ (selon l'ordre total, P_j a demandé avant) ou si P_j est actuellement en section critique :
      Ajouter P_i dans la file de P_j
      ii. Sinon :
      Répondre "C_j : P_j grant" à P_i
   (c) Un processus rentre en section critique quand il a reçu un grant de tous les autres. (C'est bien un test local.)
   (d) Quand P_i sort de section critique, il envoie le message "C_i : P_i grant" à tous les processus de sa file d'attente.

Le scénario précédent d'accès à la ressource se déroule maintenant ainsi :



P n'a pas à prévenir quiconque à sa sortie de section critique, car il n'est au courant d'aucune demande. Pas la suite, il autorisera immédiatement toutes les demandes, n'étant pas lui-même demandeur.

Lorsque Q reçoit la demande de R, il calcule que cette demande est ultérieure à la sienne. Il stocke dont la demande dans une file, et n'y répondra qu'après avoir été satisfait.

À l'inverse, lorsque R reçoit la demande de Q, il que cette demande précède la sienne et l'autorise immédiatement.

Que pensez-vous de la résistance aux pannes (p.ex. panne de machine) d'un tel algorithme ?

2.11 Mesure de complexité

Les communications étant asynchrones, il est impossible de calculer le temps d'exécution d'un algorithme distribué. On peux mesurer sa complexité en nombre de message échangés, ainsi qu'en mémoire utilisée.

Notons n le nombre de processus du système réparti. Dans l'algorithme d'exclusion mutuelle de Lamport, pour chaque section critique on envoie

– $n - 1$ messages “req”,
– au plus $n - 1$ messages “ack” (car il peuvent être facultatifs sous certaines conditions)
– puis, en sortie, $n - 1$ messages “rel”.
Soit entre $2(n - 1)$ et $3(n - 1)$ messages par section critique.

Dans l’amélioration de Ricart et Agrawala, pour chaque section critique on envoie
– $n - 1$ messages “req”,
– $n - 1$ messages “grant”.
Soit exactement $2(n - 1)$ messages par section critique.

La consommation mémoire est visiblement linéaire en n dans les deux cas.

Pour l’algorithme de Lamport, chaque processus stocke :
– une file de requêtes avec jusqu’à n entrées,
– un tableau des dates des derniers messages reçus des $n - 1$ autre processus.
Pour Ricart et Agrawala :
– une file d’attente de requêtes avec jusqu’à $n - 1$ entrées
– un tableau des autorisations reçues des $n - 1$ autre processus
– la date de sa demande d’entrée en section critique, le cas échéant

2.12 Dépendance cachée

Considérons le scénario suivant sur un système réparti à l’échelle de la planète, dans lequel deux ordinateurs “A” et “B” peuvent faire des requêtes auprès d’un serveur éloigné :
– une personne fait une requête “a” depuis l’ordinateur “A”,
– cette même personne téléphone à une autre personne pour lui demander de faire une requête “b” depuis l’ordinateur “B”.
Il est imaginable que la requête “b” parvienne au serveur avant la requête “a”. La causalité ($\rightarrow$) telle que nous l’avons définie reste respectée car l’échange téléphonique ne fait pas partie du système réparti.

Une telle situation n’est cependant pas toujours acceptable. Par exemple les requêtes pourraient être des réservations, et il serait regrettable que “b” puisse doubler “a”.

Considérons donc une relation de causalité forte “$\twoheadrightarrow$” qui prend en compte les événements pertinents de l’extérieur du système (tel que l’appel téléphonique qui sépare les deux requêtes).

Est-il possible de construire un système d’horloges C tel que la condition

$$a \twoheadrightarrow b \implies C(a) < C\langle b)$$

que nous appellerons *condition d’horloge forte*, soit vérifiée ?

Il n’est bien sûr pas possible de modifier l’extérieur du système. Par exemple si l’on voulait modifier le téléphone pour qu’il transmette les valeurs des horloges de Lamport, cela reviendrait à inclure le téléphone dans le système. Nous souhaitons trouver une solution qui fonctionne sans modifier l’extérieur du système.

Il est possible de construire un ensemble d’horloges physiques, qui, même si elles tournent plus ou moins indépendemment les unes des autres, vont satisfaire cette condition forte.

2.13 Horloges physiques

Supposons qu’on dispose d’une fonction $C_i(t)$ donnant sur les processus P_i la valeur de l’horloge au temps t. L’argument t représente le vrai temps, alors que la valeur $C_i(t)$ représente le temps mesuré par l’horloge.

Supposons de plus que l'horloge avance de façon continue, et que C_i est dérivable (sauf aux moments où on change sa valeur pour la régler).

$\frac{dC_i(t)}{dt}$ représente la vitesse ou le taux d'avancement de l'horloge au temps t. Pour une horloge parfaite, cette dérivée serait 1 : l'horloge avance aussi vite que le temps. Si la dérivée est plus grande que 1, l'horloge prend de l'avance, et à l'inverse si la dérivée est plus petite, l'horloge prend du retard. En fait, on peut raisonnablement supposer que pour toute horloge (physique) digne de ce nom on a $\frac{dC_i(t)}{dt} \approx 1$.

Plus précisément, on supposera que la dérive de toutes les horloges du système peut être majorée :

$$\exists \kappa \text{ t.q. } \forall i, \left| \frac{dC_i(t)}{dt} - 1 \right| < \kappa \tag{2.1}$$

Par exemple pour des horloges à Quartz, $\kappa \leq 10^{-6}$.

On souhaite aussi que les horloges du systèmes restent a peu près synchrones. Autrement dit, il existe majorant ε (petit !) de l'écart entre les horloges :

$$\forall i, \forall j, |C_i(t) - C_j(t)| < \varepsilon \tag{2.2}$$

Comme les vitesse des horloges diffèrent, il faudra un algorithme pour synchroniser les horloges périodiquement afin de s'assurer que la condition (2.2) reste respectée.

Quelles conditions faut-il sur κ et ε pour s'assurer que la condition d'horloge forte soit respectée ? C'est-à-dire afin que $a \twoheadrightarrow b \implies C(a) < C(b)$.

Nous savons déjà faire en sorte que $a \to b \implies C(a) < C(b)$ en utilisant le principe des horloges de Lamport. Occupons-nous donc du cas où $a \twoheadrightarrow b$ mais $a \not\to b$, c'est le cas de l'appel téléphonique de notre exemple.

Notons μ un délai tel que si a se produit au temps t, alors b se produira *après* le temps $t + \mu$. Au pire on peut borner le temps de transmission par la vitesse de la lumière avec $\mu = dist \cdot c$.

Si $a \in hist(P_i)$ et $b \in hist(P_j)$, pour respecter la condition forte, on veut donc $C_i(t) < C_j(t + \mu)$, soit $C_j(t + \mu) - C_i > 0$.

À partir de (2.1), on obtient :

$$1 - \kappa < \frac{dC_j(t)}{dt} < 1 + \kappa$$
$$1 - \kappa < \frac{C_j(t + \mu) - C_j(t)}{\mu} < 1 + \kappa$$
$$\mu(1 - \kappa) < C_j(t + \mu) - C_j(t) < \mu(1 + \kappa)$$

D'autre part (2.2) nous apprend que

$$-\epsilon < C_j(t) - C_i(t) < \epsilon$$

Donc en sommant les deux dernières équations :

$$\mu(1 - \kappa) - \epsilon < C_j(t_\mu) - C_i(t) < \mu(1 + \kappa) + \epsilon$$

Pour que la quantité $C_j(t + \mu) - C_i$ soit positive, il suffit donc que $\mu(1 - \kappa) - \epsilon \geq 0$. Autrement dit il suffit que

$$\frac{\varepsilon}{1 - \kappa} \leq \mu \tag{2.3}$$

Cette dernière contrainte, ajoutée à (2.1) et (2.2) permettra de préserver la condition d'horloge forte. Elle lie la précision de la synchronisation des horloges (ε), la précision de mesure du temps des horloge (κ), avec le temps de propagation minimal d'un message extérieur (μ).

Comment respecter (2.2) **?** Notons $d_m = t' - t$ le délai de transmission d'un message m expédié en t et reçu en t'. Le destinataire du message m ne peut pas connaître d_m, mais supposons qu'il connaisse un délais minimum $\mu_m \geq 0$ (lié par exemple à la distance entre l'expéditeur et de destinataire) tel que $\mu_m \geq d_m$. On notera $\xi_m = d_m - \mu_m$ le délai imprévisible de la communication.

Voici les règles de mise-à-jour des horloges [6] :

(IR'1) Pour tout i, si P_i ne reçoit pas de message au temps t, alors C_i continue d'avancer normalement (on rappelle que C_i est une fonction continue).

(IR'2) (a) si P_i envoie un message m au temps t, alors le message contient la date $T_m = C_i(t)$,
(b) si P_j reçoit un message m en t, alors P_j ajuste son horloge de façon atomique avec $C_j(t) \leftarrow \max(C_j(t), T_m + \mu_m)$.

Il ne reste qu'à s'assurer que des messages sont échangés assez régulièrement, pour garantir (2.2). Jusqu'à présent nous avons toujours supposé que chaque processus pouvaient envoyer des messages à tous les autres processus du système. Le résultat qui suit peut être énoncé de façon plus générale lorsque les canaux de communications entre les processus forment un graphe orienté (pas forcément complet).

Théorème 1. *Dans un graphe fortement connexe de diamètre d [7] représentant un système de communications entre des processus qui respectent les règles* **(IR'1)** *et* **(IR'2)**, *supposons que pour tout message m, $\mu_m \leq \mu$ et que $\forall t \geq t_0$:*
- *la condition* (2.1) *est respectée,*
- *il existe des constante τ et ξ telles que toutes les τ secondes, un message avec $\xi_m \geq \xi$ est envoyé sur chaque arc.*

Alors la condition (2.2) *est respectée avec $\varepsilon \approx d(2\kappa\tau + \xi)$ [8] pour tout $t > t_0 + \tau d$.*

τd représente le temps de synchronisation initiale.

2.14 Le problème de la diffusion causale

On parle de diffusion lorsqu'un message est envoyé à plusieurs destinataires à la fois.

broadcast = diffusion générale (à tous les processus)

multicast = diffusion sélective (à des processus désignés)

Nous avons déjà utilisé des diffusions générales en sections 2.9 et 2.10 : lorsque les processus envoyaient des demandes d'entrée en section critique à tous les autres.

Généralement ce mécanisme est utilisé pour partager un état ou pour chercher à maintenir la cohérence d'informations réparties (p.ex. la liste des processus souhaitant entrer en section critique).

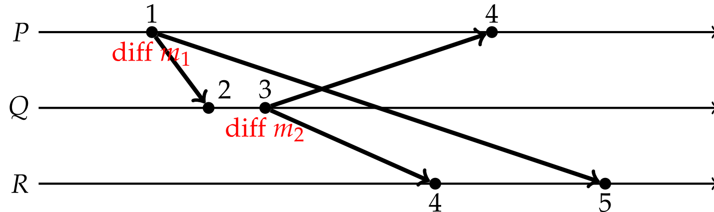
Notons $diff_i(m)$ l'événement correspondant à la diffusion (générale) d'un message m depuis le processus P_i et notons $rcpt_k(m)$ l'événement correspondant à la réception de ce message par le processus P_j.

La *diffusion causale* est un mécanisme qui garantit que deux diffusion causalement dépendantes seront reçues dans le bon ordre. Autrement dit :

6. Notez que le temps physique t apparaît dans la définition des règles mais n'est pas utilisé en pratique.
7. Cela signifie que deux sommets du graphe peuvent être reliés en suivant au plus d arcs.
8. Cette approximation suppose que $\mu + \xi \ll \tau$.

$$\forall m, m', \forall i, j, \, diff_i(m) \to diff_j(m') \implies \forall k, rcpt_k(m) \to rcpt_k(m')$$

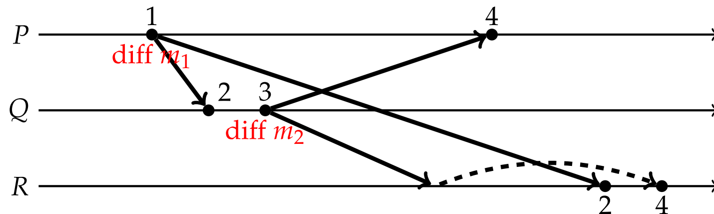
Voici deux diffusions dont l'enchaînement ne respecte pas la causalité



Dans l'exemple ci-dessus, la diffusion du message m_1 précède causalement la diffusion du message m_2, pourtant la réception par le processus R ne respecte pas cet ordre.

Imaginons que les trois processus P, Q et R soient des réplicats d'une base de données répartie représentant des comptes bancaires. Les trois réplicats peuvent être sur trois site différents, et les opérations effectuées sur n'importe quel site doivent être répercutées sur les autres sites pour maintenir la cohérence des donnés. Si le message m_1 crédite un compte ("`+= 100`") alors que le message m_2 ajoute les intérêts annuels ("`*= 1.02`"), on conçoit que ces opérations doivent absolument être effectuées dans le même ordre sur tous les sites. Sans cela les valeurs calculées par chaque site seraient différentes.

Un algorithme de diffusion causale (qui serait donné par le middleware) doit éviter ce scénario sur le processus R en réordonnant les messages reçus. Lors de la réception de m_2, il doit trouver (nous verrons comment plus tard) qu'une diffusion plus ancienne est en route depuis P, puis garder m_2 en attente jusqu'à ce que m_1 ait été délivré.



Un utilisant les horloges de Lamport, une technique permettant de décider si m_2 peut-être délivrée immédiatement serait d'utiliser des canaux FIFO et de s'assurer qu'on a reçu de la part de tous les autres processus un message avec une date ultérieure à la diffusion reçue. Il faudra évidement provoquer des échanges de messages lorsque ce n'est pas le cas.

Ne nous attardons pas sur cette technique, car les horloges *vectorielles* permettent de résoudre ce problème de façon plus élégante.