CHAPITRE 10

SYNCHRONISATION PAR MESSAGES

Plan

COMMUNICATION PAR MESSAGE
INVOCATION À DISTANCE
programmation des paradigmes
avec des tâches Ada comme serveurs
MESSAGES ASYNCHRONES
dans la famille POSIX, UNIX, LINUX
SYNCHRONISATION RÉPARTIE
introduction aux systèmes répartis
exclusion mutuelle répartie

Manuel 10

Synchronisation par messages

1. Communication par messages

La concurrence de processus peut se produire dans deux classes d'architecture de systèmes : systèmes centralisés ou systèmes répartis. Nous avons étudié aux chapitres 8 et 9 le cas des systèmes centralisés, c'est à dire ceux où la concurrence des processus et le partage des ressources sont contrôlés en utilisant des données communes en accès direct. On examine maintenant le cas des systèmes répartis communiquant uniquement par messages entre processus concurrents. L'architecture matérielle qui supporte ce type de systèmes est une architecture distribuée autour d'un réseau local ("local area network"), ou utilisant un réseau général ("wide area network").

On suppose, pour simplifier l'étude et rester au niveau logique, et ne pas avoir à se préoccuper de l'allocation des processeurs aux processus, qu'il y a autant de processeurs (ou de sites du réseau, chaque site étant monoprocesseur) que de processus.

1.1. Mode Message synchrone entre processus ("rendez-vous" simple)

L'émetteur et le récepteur sont tous deux en même temps (d'où le terme synchrone) dans l'état de communication. L'émetteur émet son message et attend la fin de réception par le récepteur. Un récepteur qui attend un message est bloqué jusqu'à son arrivée. On a un fort couplage temporel entre les deux processus. On parle de rendez-vous simple (simple, car on ne transmet qu'un seul message pendant le rendez-vous).

1.2. Mode Message asynchrone entre processus

L'émetteur et le récepteur traitent un message donné à des instants différents (d'où le terme asynchrone). L'émetteur émet son message et n'attend pas la fin de réception par le récepteur. Un récepteur qui attend un message n'est pas nécessairement bloqué jusqu'à son

arrivée. On applique le paradigme producteur consommateur avec un tampon à l'émission et un tampon de réception. On a une indépendance temporelle entre les deux processus. Mais dans ce schéma, il est plus compliqué de savoir si un message a été bien reçu et comment cette réception se place par rapport aux actions faites par l'émetteur après l'envoi, ce qui rend difficile le traitement et la correction des erreurs.

1.3. Mode Invocation à distance entre processus ("rendez-vous" étendu)

Un processus appelant demande l'exécution d'une procédure qui est un composant du processus appelé. Ce mode comprend un premier message qui correspond au passage des paramètres et un second message qui correspond au passage des résultats. Comme si la procédure était chez l'appelant, celui-ci attend le passage des résultats avant de poursuivre.

L'écriture de l'appel ressemble à un appel de procédure. Il peut ne pas y avoir de résultat, mais l'appelant attend toujours la fin de l'exécution par l'appelé. Ceci apporte une sémantique précise, celle de l'appel de procédure.

L'appelant émet son message et attend la fin du traitement et la réponse de l'appelé. On a un fort couplage temporel entre les deux processus pendant l'exécution de l'appel de procédure. On parle de rendez-vous étendu.

Ce mode est différent de l'appel de procédure à distance qui est à un niveau physique (on sait que la procédure appelée à distance est sur un autre site) et non à un niveau logique (alors que l'objet procédure invoqué à distance est nécessairement un composant d'un processus connu)

1.4. Analogies et implantation des modes de communication

Une analogie peut être faite pour ces trois modes avec envoyer un fax (synchrone), poster une lettre (asynchrone), et téléphoner (invocation à distance);

La communication par messages synchrones a été utilisée dans les langages CSP et Occam et dans des architectures comme les transputeurs. C'est un mode qu'on trouve surtout dans les applications scientifiques réparties.

La communication par messages asynchrones est le mode privilégié fourni par les interfaces systèmes ou les interfaces réseaux. Son implantation en architecture centralisée utilise des tampons du noyau et le paradigme producteur consommateur. Son implantation dans les systèmes répartis (Chorus, Mach, Amoeba,...) utilise les protocoles réseaux (datagrammes, sockets,...).

La communication par invocation à distance est une abstraction du niveau des langages de programmation. La comparaison avec l'appel de procédure distante fait ressortir les principales différences suivantes :

construction langage opposée à construction réseaux
appel d'un processus (objet actif) opposé à appel de procédure (objet passif)
exécution par un processus connu opposé à exécution par un courtier inconnu

2. Invocation à distance entre tâches Ada

Dans le langage Ada, la communication de base entre tâches peut se faire de deux façons, soit par le partage d'un objet protégé, soit par invocation à distance entre tâches (appelée le "rendez-vous" Ada). Nous avons rattaché le premier cas aux méthodes de synchronisation par mémoire commune. Nous traitons le second cas comme exemple de synchronisation par messages avec des processus serveurs.

On utilise la terminologie suivante (qui n'est pas dans le langage Ada) :

- client : c'est la tâche appelante
- serveur : c'est la tâche appelée

Le "rendez-vous" entre deux tâches est la situation où le client a envoyé une requête et où le serveur traite la requête ou est prêt à la traiter.

2.1. Rendez-vous entre tâches Ada

Le rendez-vous Ada est dissymétrique :

- le client fait une invocation à distance programmée comme un appel de procédure ; le serveur sert une requête arrivée sur un point d'entrée et programme par "accept" son accord pour traiter la prochaine requête présente.
- le serveur est connu de tous les clients ; le serveur ne connaît pas le client dont il sert la requête et ne peut pas choisir un client particulier à servir.

2.1.1. *Les éléments du serveur (tâche appelée)*

Le serveur déclare des points d'appel que le client doit désigner dans son appel et les types des paramètres d'appel, données "in" et résultats "out", qui doivent accompagner l'appel.

Le serveur accepte de traiter une requête déposée en un point d'entrée par "accept".

Le serveur peut mettre une condition (on dit une garde) à cette acceptation : "when condition => accept". La condition est une expression booléenne avec des données du serveur, donc elle ne peut contenir ni le nom du client, ni un paramètre de l'appel.

Le serveur doit décider de la fin du rendez-vous, ce qui doit entraîner le réveil du client :

- la lecture des paramètres, le traitement et l'envoi de la réponse doivent être faits avant de réveiller le client ; pour cela on les encadre par "do".... "end;" et c'est l'exécution du "end;" qui entraîne le réveil du client. La requête peut entraîner des actions de gestion locale au serveur qui peuvent se dérouler après le "end;", donc après le réveil du client.
- si la requête est sans paramètres et si le client peut être réveillé dès que la requête est lue par le serveur, on omet le "do"..."end;" et c'est l'exécution de l'"accept" qui déclenche le réveil.
- si la requête acceptée ne peut être menée à terme, on peut ne pas réveiller le client et remettre la requête en attente sur un point d'entrée (le même, ou un autre) par l'instruction "requeue" ; "requeue" ne déclenche pas le réveil du client. Cela permet de traiter une requête en plusieurs étapes : analyse de la demande qui figure dans les paramètres (quantité demandée, identité du client), essai de satisfaire cette demande et si impossible, mise en attente, ré-essais successifs jusqu'à ce que ce soit possible.

Le serveur ne sait pas s'il y a une requête au point d'entrée, donc il peut être bloqué s'il n'y a pas d'appel de client en attente. Il ne sera réveillé qu'à l'arrivée d'une requête.

Si on veut que le serveur attende sur plusieurs entrées, on les encadre par une clause d'attente sélective : "select"... "end select". Le serveur traite alors l'une des requêtes possibles et une seulement, quel que soit le nombre des requêtes concernées. Si plusieurs requêtes sont exécutables (requêtes présentes et conditions des gardes à vrai) le choix de la requête à traiter est quelconque (c'est l'indéterminisme du choix). S'il n'y a aucune requête exécutable, le serveur attend. C'est ainsi qu'on bloque un serveur cyclique pour attendre les demandes de ses clients.

On peut ajouter une autre clause d'attente sur l'attente sélective :

- "terminate" autorise le système à détruire le serveur si tous ses clients potentiels ont terminé leur exécution (le moteur d'exécution des tâches Ada - le "run time" - sait le déterminer en ligne grâce à la structure de bloc du langage
- "delay D" ou "delay until E" arrête l'attente au bout de D unités de temps ou à la date E
- "else" supprime l'attente s'il n'y a pas de requête exécutable.

2.1.2. *Les éléments du client (tâche appelante)*

Le client fait un appel procédural à destination d'un serveur qu'il nomme et dont il spécifie aussi un point d'appel. Il fournit les paramètres effectifs pour l'appel.

Si on veut que le client n'attende pas le serveur si celui-ci n'est pas au rendez-vous, l'appel du client peut comporter une clause de rendez-vous conditionnel :

"select...; else...end select"

Si on veut limiter l'attente du client avant le rendez-vous, par un délai ou une date, on utilise : "select...; or delay [until]...end select"

2.1.3. *Un exemple*

Le processus principal crée deux fils serveurs d'impression et le père les utilise comme un client.

Chaque fils a un point d'appel entry Start(Id : Integer);

Quand il accepte une requête, il l'exécute en visualisant la valeur du paramètre Id et c'est après l'affichage qu'il libère le client par "end Start;"

Le père appelle son premier serveur Fils1 à son point d'appel Start() et attend la fin de cette requête avant de lancer la seconde au second serveur. On est sûr ainsi que les visualisations sont sérialisées dans l'ordre indiqué. (on n'a qu'une seule console)

Exercice : On associe à chaque fils une console qui lui est propre. Que faudrait-il faire pour laisser indéterminé l'ordre de visualisation des lignes de chaque fils?

2.2. Programmation des paradigmes avec des tâches serveurs

Chaque paradigme est mis en oeuvre par une tâche serveur invocable en concurrence par les autres tâches. Tous les messages passent, au moins au début de la coopération, par le serveur.

2.2.1. *Exclusion mutuelle*

Tout serveur gère les données internes qui lui appartiennent et les requêtes sont sérialisées. Il suffit de préparer des points d'appel pour chaque type d'action à faire sur les données internes. Et les clients appelant ces points d'appel sont servis les uns après les autres. L'exclusion mutuelle est structurelle pour ces données internes au serveur.

Le serveur d'exclusion mutuelle peut aussi servir à fournir des mécanismes de coopération si l'exclusion mutuelle ne porte pas seulement sur des données gérées par un serveur. On réalise un sémaphore binaire ou un verrou, utilisable pour contrôler l'accès à une section critique quelconque. On utilise une tâche serveur Mutex qui a un point d'appel appelé P, un autre appelé V. Cette tâche accepte les requêtes de ses clients seulement dans l'ordre immuable P, puis V, puis P, puis P,...C'est la suite [P V]*. On l'obtient par la programmation du cycle :

"loop accept P; accept V; end loop;"

C'est le corps du serveur.

Si on veut qu'il arrête de boucler indéfiniment quand il n'aura plus de clients potentiels, on ajoute la clause d'attente sélective avec terminaison. On note que cette clause ne joue que sur la paire "accept P; accept V; ". C'est à dire qu'un serveur qui vient de servir un client qui a invoqué "Mutex.P;" ne peut pas être détruit car il est dans l'instruction "select" où il attend un rendez-vous avec un client qui invoque "Mutex.V;".

2.2.2. *Producteur Consommateur*

Pour implanter le paradigme producteur consommateur, on crée un serveur qui contient un tampon de stockage et deux points d'appel, l'un pour obtenir le dépôt, l'autre pour obtenir le retrait d'un message. Comme tout serveur ne traite qu'une requête à la fois, il n'y a donc jamais de parallélisme entre dépôt et retrait. Le serveur libère le client dès que le message a été déposé ou retiré, et la gestion de l'index et du nombre est faite par le serveur, après cette libération.

2.2.3. *Rendez-vous symétrique synchrone*

Pour construire un rendez-vous synchrone, on utilise un serveur qui contient deux points d'appel, l'un pour émettre le message, l'autre pour recevoir le message. Le serveur attend l'émetteur, puis quand il est au rendez-vous, il attend le récepteur sans pour autant libérer l'émetteur. Quand l'émetteur et le récepteur sont tous deux au rendez-vous avec le serveur, celui-ci peut passer le message de l'un à l'autre, du paramètre effectif "in" de l'émetteur au paramètre effectif "out" du récepteur, sans utiliser de recopie dans le serveur. Puis les deux clients sont libérés. Si le récepteur ne se manifeste pas, l'émetteur reste bloqué. La clause "terminate" ne joue que si la paire de processus s'est manifestée.

2.2.4. *Repas des philosophes*

Le serveur proposé contient deux points d'appel externes, l'un pour acquérir les deux baguettes d'un philosophe x ("Demander()"), l'autre pour les rendre ("Conclure()"), et un tableau de booléens repérant la disponibilité des baguettes. Il implante la politique qui ne sert un philosophe que lorsque les baguettes sont toutes deux disponibles et qui sinon le met en attente sur un point d'appel privé "Re_Demander()". Quand des baguettes sont rendues, on note le nombre de requêtes qui sont en attente sur ce point d'appel privé et qui doivent être réexaminées. Cette mise à jour du nombre peut rendre exécutables les requêtes en attente, car la garde du point

d'appel privé est vraie si le nombre est positif. À chaque requête exécutée par Re_Demander(), on décrémente le nombre initial, ce qui permet le réexamen de toutes les requêtes. Si les deux baguettes demandées par une requête ne sont pas disponibles, celle-ci est remise en attente sur le point d'appel privé "Re_Demander()".

2.2.5. *Allocation de ressource*

Pour allouer les ressources une à une, le serveur contient deux points d'appel, l'un pour prendre une ressource, l'autre pour rendre une ressource. Il comprend une variable entière qui indique le nombre de ressources encore libres, et qui sert à créer une garde validant l'acceptation d'une requête pour prendre une ressource.

Pour allouer plusieurs ressources à la fois, le serveur contient deux points d'appel, l'un pour prendre X ressources, l'autre pour rendre X ressources. Il comprend une variable entière qui indique le nombre de ressources encore libres. On accepte des requêtes sur Prendre(X : in Integer) tant qu'il y a des ressources libres. Quand il n'y en a plus assez, on met la requête dans le point d'appel Finir et on interdit toute nouvelle acceptation de requête sur Prendre. Quand des ressources sont rendues, on les ajoute au nombre des ressources libres et s'il y en a suffisamment, on réveille le processus client bloqué sur Finir. On obtient ainsi un service à l'ancienneté ("FIFO") car toute requête non servie interdit de servir les suivantes.

2.3. Invocation conditionnelle ou limitée dans le temps

L'exemple d'attente sélective coté serveur montre comment programmer le serveur pour qu'il accepte un signal et qu'il note qu'il a été reçu et pour que, lorsque l'attente atteint 30 unités de temps, l'attente du signal cesse, qu'une action de sauvegarde soit prise et que le serveur note que le signal n'a pas été reçu.

Dans l'exemple du rendez-vous conditionnel, s'il n'y a pas de requête en attente, on l'indique et le serveur ne reste pas au rendez-vous.

L'exemple de demande de rendez-vous conditionnel montre comment le client peut noter si le serveur est ou non au rendez-vous et s'il n'y est pas, annuler son invocation.

L'exemple d'appel d'entrée à attente limitée dans le temps montre comment annuler l'appel si au bout de 20 unités de temps le rendez-vous n'a toujours pas eu lieu.

2.4. Simulation des sémaphores par des tâches

On peut simuler un type de sémaphore booléen et l'utiliser comme sémaphore d'exclusion mutuelle. On peut aussi simuler un type de sémaphore avec compte.

La simulation de sémaphores par des tâches a l'avantage de fournir des points de synchronisation lors des rendez-vous, ce qui force alors la mise en cohérence de variables partagées entre les tâches. (voir Manuel 8 au 3.4.)

Messages asynchrones dans la famille Posix-Unix-Linux

On présente les mécanismes de communications inter-processus dans les systèmes de la famille Posix-Unix-Linux. Ils sont fondés autour de la notion de tube (ou "pipe"), canal unidirectionnel en mode flot d'octets et dans lequel un processus peut écrire à un bout et lire à l'autre bout. Une table de descripteurs de tubes est associée à tout processus. Un processus créé par un "fork" hérite des tubes de son père et y a accès au même titre que son père. C'est ainsi que se fait le partage des tubes. Il faut ensuite que le père et le fils se donnent des conventions d'utilisation du tube pour que l'un y écrive et l'autre y lise. Avec deux tubes, on met en place une relation client serveur qui permet au père client (par exemple) d'envoyer des requêtes dans un tube et de recevoir les réponses dans un autre.

Ce mécanisme d'héritage permet à des processus d'une même famille de communiquer. Il faut que les processus existent. Pour permettre à des processus de familles indépendantes de communiquer entre eux, on a ajouté des tubes nommés qui sont des fichiers gérés comme un tube. Ils sont donc créés et désignés et mémorisés dans la bibliothèque des fichiers du système. Un processus accède à un tube nommé comme à un fichier et doit l'ouvrir et le fermer. L'avantage est que le tube nommé continue d'exister après la terminaison du processus. Il est

public. La destruction du tube nommé doit être explicite. Le tube nommé peut être utilisé pour créer des boites aux lettres permanentes pour les clients et le serveur

La suite d'octet déposée dans un tube est de longueur quelconque et est marquée par un délimiteur conventionnel "/n". La suite lue n'est pas nécessairement de même taille. On gère des flots d'octets et non des structures. Pour pouvoir gérer automatiquement des messages structurés et typés, par exemple retirer les messages en fonction de leur type, on a introduit, dans Unix System V, des files de messages qui sont gérés comme des fichiers et désignés par un identificateur unique qui est son chemin d'accès dans le catalogue des fichiers (voir en annexe du cours).

Pour information et pour faire le lien entre Unix et le cours, on indique aussi ce qu'est la notion de groupe de sémaphores présente dans Unix et comment on peut l'utiliser pour définir les opérations P, V et E0 sur les sémaphores vues dans le cours.

4. Synchronisation dans les systèmes répartis asynchrones

4.1. Systèmes répartis asynchrones

Un système réparti asynchrone est un ensemble de sites interconnectés par un réseau de communication qui véhicule des messages asynchrones. On suppose que les sites communiquent tous entre eux.

Un système réparti asynchrone a des propriétés (ou des absences de contraintes) caractéristiques :

pas de mémoire, pas d'horloge commune entre sites, pas de borne maximale du temps de transfert d'un message, pas de vitesse minimale pour un processeur.

Ces absences de contraintes engendrent des difficultés particulières qui compliquent le contrôle des systèmes répartis asynchrones.

Deux sites qui observent un ensemble d'événements du système réparti peuvent les percevoir avec un ordre différent et leur attribuer des dates qui ne sont pas les mêmes.

Donc les sites ont des visions différentes de l'état des ressources du système et n'ont pas de repère commun au moment où il faut prendre sur chaque site une décision cohérente avec la décision prise sur les autres sites (entrer en exclusion mutuelle, élire un site directeur, connaître la taille d'un groupe, lancer un calcul coordonné, terminer un calcul réparti).

Le grand nombre de sites d'un système réparti a comme conséquence un risque accru de défaillance. La défaillance ou l'absence d'un site n'est plus un événement rare et on ne peut pas l'ignorer. Il devient important de faire des hypothèses précises sur les défaillances possibles, d'envisager des modules détecteurs de défaillance et de développer des méthodes utilisant la suspicion de défaillance.

En effet, un processus Pi sans réponse de Pj ne peut savoir si Pj prépare sa réponse (Pj serait lent), si la réponse est faite et en route (le message serait lent) ou si Pj ne peut pas répondre (Pj serait en panne). A partir de quel délai peut-on suspecter Pj d'être en panne? Cette détection de défaillance n'est pas fiable à 100%, mais certaines hypothèses permettent néanmoins de s'en servir utilement dans un modèle de système réparti appelé quasi asynchrone.

Un exemple de communications pour une mise à jour des copies d'un objet répliqué sur 4 sites montre que les sites ont des visions différentes des répliques, même s'ils partent tous d'une même valeur initiale.

Plus généralement, il existe un théorème d'impossibilité démontré en 1985 par Fischer, Lynch et Paterson pour le consensus (choix réparti d'une valeur commune):

Il est impossible de concevoir un protocole déterministe pour résoudre le problème du consensus dans un système réparti asynchrone dans lequel (seulement) un processus peut être défaillant par panne franche.

Intuitivement ceci est dû à l'impossibilité de distinguer un processus défaillant d'un processus extrêmement lent.

4.2. Causalité due aux messages et datation par horloge logique

La réception d'un message permet d'observer la relation de causalité physique élémentaire entre l'émission et la réception d'un message : l'émission précède la réception (en anglais "happened before").

L. Lamport a combiné les relations de causalité entre les événements d'émission et de réception des messages observés lors d'une application répartie, et les relations d'ordre locales à chaque site monoprocesseur. Cela donne un ordre partiel pour l'ensemble des événements du système. Dans cette relation d'ordre partiel, des événements demeurent incomparables.

Pour pouvoir les comparer, il faut introduire un ordre total qui respecte l'ordre partiel entre deux événements, partout où il existe et qui classe arbitrairement, mais de la même façon sur chaque site où intervient le classement, les événements incomparables par l'ordre partiel.

Pour mettre en oeuvre cet ordre partiel et cet ordre total, L. Lamport a introduit des horloges logiques, une par site, qui servent à dater sur chaque site les événements qui s'y produisent. Chaque événement daté incrémente l'horloge locale ; et l'événement réception d'un message est utilisé pour remettre à jour un site qui ne serait plus en relation de causalité avec le site émetteur du message. Pour cela chaque message est estampillé sur le site émetteur avec la date donnée à l'événement émission du message. Sur le site récepteur et avec l'horloge locale à ce récepteur, il faut que la date de réception soit postérieure à la date d'émission véhiculée par le message. Cela peut entraîner une remise à l'heure de l'horloge du récepteur.

Cette datation peut donner la même valeur à des événements incomparables. On suppose que tous les sites sont repérés par une numérotation unique. Pour obtenir un ordre total, on utilise cette numérotation.

4.3. Exclusion mutuelle répartie

L'algorithme d'exclusion mutuelle introduit par L. Lamport en 1978 suppose que le nombre de sites est connu, que le réseau est connexe et que les canaux sont FIFO. Les événements du système sont classés par un ordre total obtenu avec une datation par horloges logiques.

Le principe de l'algorithme repose sur la connaissance mutuelle acquise par chaque site de façon répartie et par la création de files d'attentes dupliquées.

Quand un site veut entrer en section critique, il demande l'autorisation à tous les sites et il n'entre en section critique que lorsqu'il sait que sa demande est la plus ancienne. Il a cette information de deux façons, soit parce qu'il a connaissance des dates de demandes d'entrée en section critique des autres sites, soit, pour les sites qui ne demandent pas à entrer en section critique, parce qu'il l'a appris et qu'il sait qu'ils ne feront pas dans le futur de demande qui pourrait être datée avant la sienne. Pour obtenir ce deuxième type d'information, chaque site répond à une demande d'entrée en section critique par un message d'acquittement. Ce message peut être utilisé de façon significative car les canaux sont FIFO et il ne pourra pas être dépassé par une demande d'entrée en section critique. Quand un site quitte la section critique, il averti tous les sites pour que son successeur puisse se reconnaître.

Chaque site établit une veille pour savoir s'il peut entrer en section critique. Il utilise un ensemble de N messages significatifs, un par site du système. En recevant des messages, il met éventuellement à jour cet ensemble et il n'entre en section critique que lorsqu'il a un message de chaque site (il doit avoir connaissance des demandes ou non demandes de tous) et que son message est la plus ancien message de cet ensemble.

Cet algorithme engendre 3(N -1) messages. Il se termine quand tous les messages sont arrivés. Dans un système réparti asynchrone pur, l'absence de majorant connu sur le temps de transfert des messages (même dans le cas où les sites ne tombent pas en panne) ne permet pas de garantir la terminaison de l'algorithme. Pratiquement un majorant existe pendant le déroulement de l'algorithme ; un détecteur de défaillance, construit avec des délais de garde permet de vérifier son existence.