Les systèmes distribués

1. Définition d'un système distribué

Comme toujours, en informatique, chaque terme possède plusieurs définitions. Les termes « réparti » et « distribué » sont couramment employés, pour désigner les mêmes concepts fondamentaux.

Voici quelques définitions qui permettent de situer le contexte très particulier du monde de l'informatique distribuée.

Définition générale

Un système distribué est un système dont l'état global n'est pas directement observable. Son comportement est déterminé par des algorithmes explicitement conçus pour être exécutés en parallélisme asynchrone (sans horloge physique commune) par plusieurs calculateurs, chacun n'ayant qu'une vue partielle et retardée de l'état global courant.

Certains étendent cette définition de la façon suivante : « Un système distribué (aussi appelé « système réparti ») est constitué d'une collection de calculateurs qui ne partagent ni leurs mémoires physiques ni leurs horloges ».

Par contre, chaque calculateur, d'un système distribué, possède son propre processeur, de la mémoire locale ainsi qu'une horloge physique.

Les processeurs d'un système réparti communiquent entre eux par des voies de communication qui peuvent varier en nombre et en distance. Ils peuvent se trouver sous différentes formes et posséder différentes fonctionnalités (microprocesseur, microcontrôleur, processeur parallèle, station de travail …).

Dans la suite de ce cours nous emploierons le terme « processeur » ou CPU pour désigner un calculateur complet (une unité de calcul associée à de la mémoire et divers périphériques de gestion des entrées/sorties).

1.1 Les concepts matériels d'un système distribué

En fait, un système distribué est, par définition, un système multiprocesseurs. Cette structure se rencontre sous deux formes distinctes :

- la structure a processeurs « fortement couplés »
- la structure a processeurs « faiblement couplés »

1.1.1 Systèmes fortement couplés

La structure a processeurs **« fortement couplés »** est une variante très utilisée dans l'univers des systèmes temps réel. Dans cette architecture, les processeurs partagent un bus d'adresses et un bus de données leur permettant d'échanger rapidement des messages. Du fait de ce partage direct de la mémoire entre les processeurs, cette structure s'éloigne légèrement de la définition que nous venons de donner d'un système distribué. De plus, cette structure utilise souvent des matériels et logiciels spécifiques. C'est pour ces raisons, que nous ne nous attarderons pas sur la description des mécanismes de distribution de traitements dans une telle configuration.

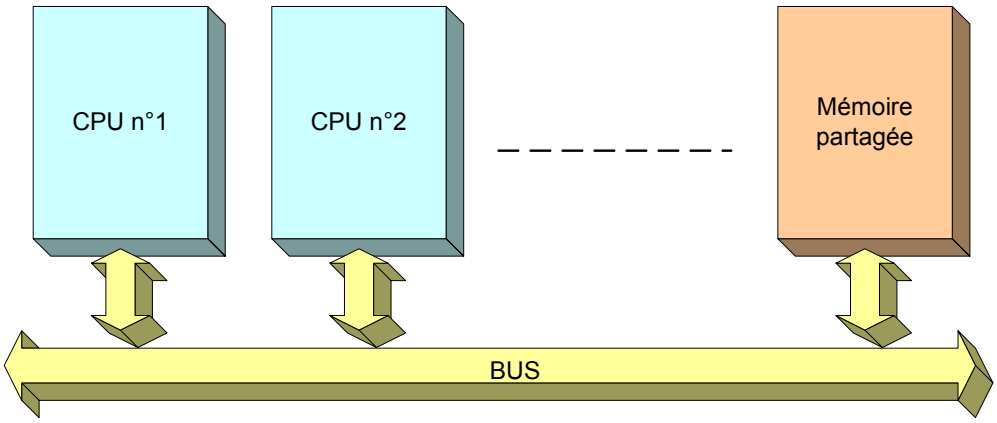


Figure 1.1.1-1 système fortement couplé

1.1.2 Systèmes faiblement couplés

La structure a processeurs **« faiblement couplés »** interconnecte un ensemble de processeurs communiquant entre eux au travers d'un réseau. Ce réseau peut être utilisé pour véhiculer des données d'une ou plusieurs applications ne communiquant pas nécessairement entre elles.

Un système distribué permet à des utilisateurs d'accéder à ses diverses ressources. L'accès à une ressource partagée permet d'accroître la vitesse de calcul et tend à améliorer la disponibilité ainsi que la stabilité des données.

Un système de fichier réparti (ou distribué) permet de disperser les utilisateurs, les serveurs ainsi que les périphériques de stockage de l'information sur les différents sites d'un système distribué. Au lieu d'un seul système central de données, on trouve plusieurs unités de stockage indépendantes.

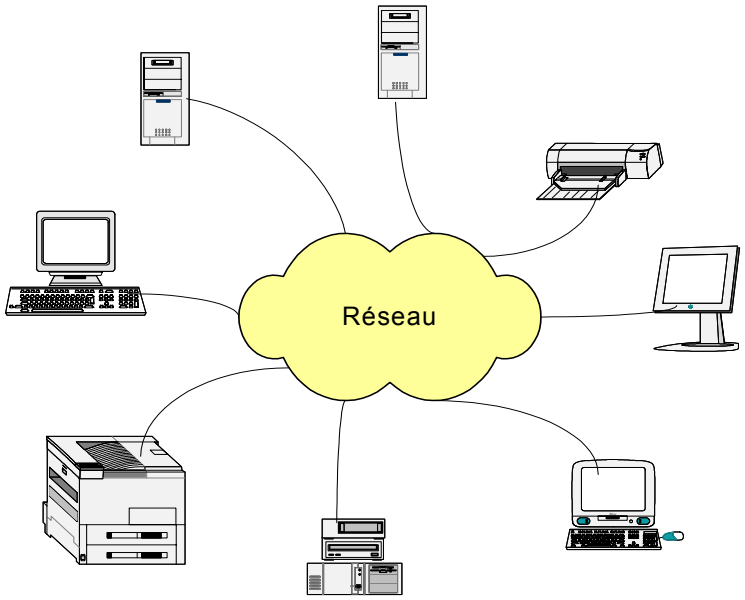


Figure 1.1.2-1 système faiblement couplé

1.2 Les concepts logiciels d'un système distribué

Nous avons vu qu'un système distribué est constitué de plusieurs processeurs faiblement couplés. Chacune des CPUs est prise en charge par un système d'exploitation local.

Certains constructeurs proposent des systèmes d'exploitation appelés « Systèmes d'Exploitation Distribués ». Dans ce cas, l'OS lui-même tient compte de l'éloignement géographique des différentes CPU et ressources associées et offre des outils spécifiques permettant de rendre transparents les aspects réseau.

La plupart des systèmes répartis actuels reposent sur un système d'exploitation « standard » auquel on ajoute des fonctionnalités permettant de déployer une solution distribuée. Dans ce cas, la gestion des aspects spécifiques à cette architecture est souvent laissée aux soins du concepteur de l'application distribuée.

C'est pour ces deux raisons que, dans ce cours, nous parlerons de « systèmes distribués » plutôt que de « systèmes d'exploitations distribués ».


Dans tous les cas, un système distribué doit fournir des mécanismes permettant la synchronisation des processus, la gestion des communications interprocessus, la solution au problème d'interblocage ainsi que la gestion de nombreux autres problèmes inexistants dans un système centralisé (tel que l'unicité des numéros de processus, la gestion des priorités entre processus, …).

Du fait de la répartition géographique des processeurs et de la multiplicité des matériels mis en oeuvre, les systèmes distribués sont souvent associés au problème de la **tolérance aux pannes**.

Il est évident qu'un système d'exploitation multitâche supportant la communication interprocessus par échange de messages se prête plus facilement au déploiement d'un système distribué. Par exemple, MS-DOS est difficilement intégrable dans un système distribué car son noyau est « monotâche » et n'offre aucun support de communication par messages.

Certains systèmes sont capables de gérer de la distribution dans des milieux hétérogènes.

Avec l'apparition des technologies « Objets », les systèmes distribués se sont tournés tout naturellement vers ces concepts qui permettent une distribution d'objets complexes sur un réseau (CORBA, RMI, DCE …).

1.2.1 Terminologie

Du point de vue d'un processeur spécifique dans un système réparti, le reste des processeurs ainsi que leurs ressources respectives seront dits « éloignés » ou « distants » tandis que ses propres ressources seront dites « locales ».


Les différents processeurs d'un système distribué sont identifiés au moyen d'un de différents noms (nœud, site, ordinateur, machine, hôte …) selon le contexte dans lequel ils sont mentionnés.

Dans la suite de ce cours, nous utiliserons essentiellement le terme « site » pour indiquer un ensemble d'ordinateurs et le terme « hôte » pour identifier un système spécifique au sein d'un site.

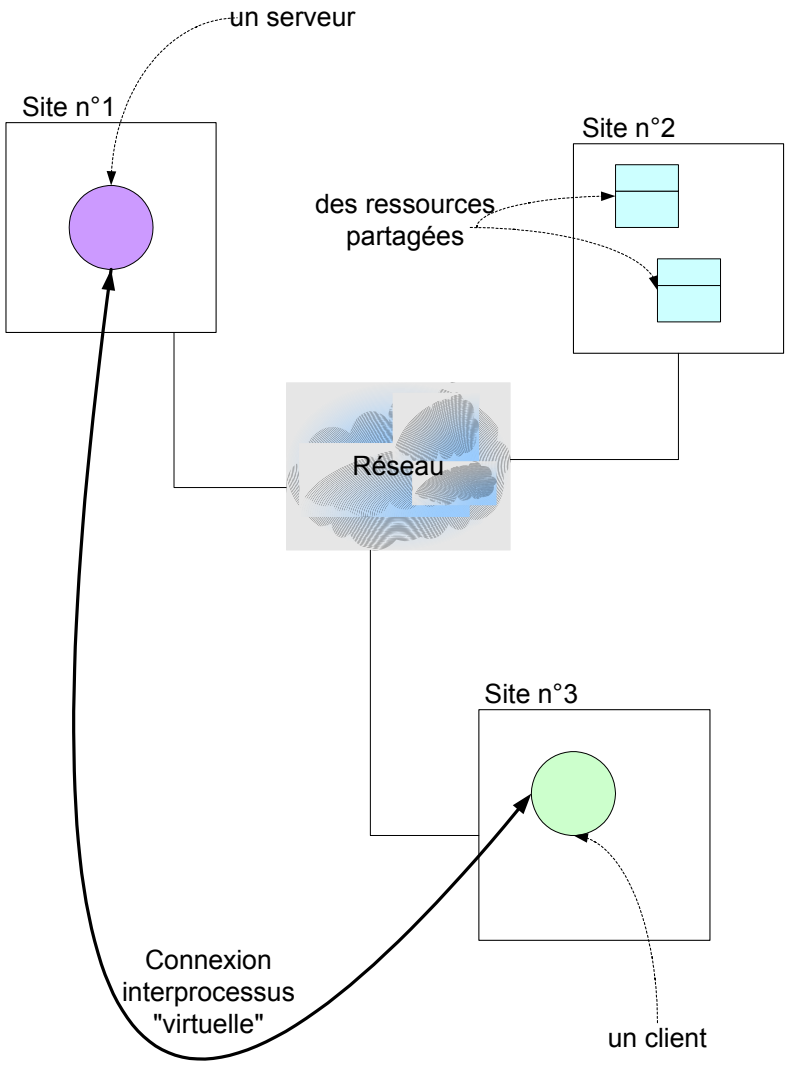


Figure 1.2.1-1 : représentation d'un système distribué

1.2.1.1 Les systèmes distribués et le « temps réel »

1.2.1.1.1 Rappels sur le temps réel

ARIAL-BOURGNE : « Un système fonctionne en temps réel s'il est capable d'adsorber toutes les informations d'entrée sans qu'elles soient trop vieilles pour l'intérêt qu'elles présentent, et par ailleurs, de réagir à celles-ci suffisamment vite pour que cette réaction ait un sens ».

B. GIRARD et G. MICHEL (Les langages évolués pour le temps réel) « Par fonctionnement en temps réel, on entend généralement une exécution de programme qui satisfait aux contraintes de temps ».

J.P. POUGET (Ordonnancement en temps réel) : « Le temps réel est le pilotage à tout instant d'un système évolutif dans le temps ».

SENOUILLET : « Le concept de temps réel s'applique aux organes qui sont inclus dans les processus dont les données et les résultats sont relatifs à des événements en cours ».

En résumé, nous pouvons définir un système temps réel comme un système qui est capable de réagir à un stimuli, dans un temps borné, et ce quel que soit l'activité courante de la machine. Les chemins de données sont tous bornés dans le temps et le système garanti l'ordre chronologique des évènements et la « priorisation » constante dans l'exécution des processus.

1.2.1.1.2 Les contraintes spécifiques des systèmes distribués temps réel

1.2.1.1.2.1 Les flux de données

Les flux de données doivent permettre l'acheminement de l'ensemble des données entre le processus et les I/O. Le temps de prise en compte de ce flux dépend des paramètres suivants :

- le volume de données à transférer,
- le temps d'acquisition des données en entrée,
- le temps de génération des données en sortie.

Dans le cas ou les données transitent par un réseau de communication, il faut ajouter les paramètres suivants :

- le temps de transfert des données en entrée,
- le temps de transfert des données en sortie,
- le débit moyen du réseau,

le débit instantané du réseau.

1.2.1.1.3 Les flux de contrôles

Ils doivent permettre la gestion des fonctions. Ils circulent souvent sur le même support physique que les données.

De ce fait, ils sont générateurs de volumes de transfert qui viennent augmenter le débit initial du flux de données.

1.2.1.1.4 les contraintes temporelles

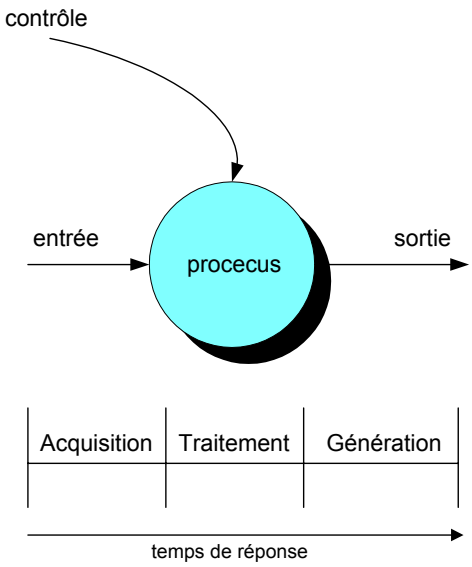
Un des problèmes du temps réel, c'est de pouvoir véhiculer, dans le temps imparti, les flots de données et les flots de contrôles.

Le deuxième problème, c'est de disposer de la puissance de traitement nécessaire pour traiter les processus.

Il faut donc trouver un compromis entre :

d'une part, des gros volumes de données à transférer et une unique CPU de grosse puissance,

et d'autre part, plusieurs CPU de taille plus réduite et un volume de données moindre du fait du traitement des données à la source.



1.2.1.1.5 La conception d'un système distribué temps réel

La conception d'un système distribué ayant des contraintes impérieuses de temps est semblable à la conception d'un système distribué fonctionnant en temps partagé (« time sharing »).

Il est évident qu'un système distribué ne peut être qualifié de temps réel que si l'ensemble des entités intervenant possèdent des caractéristiques de respect des contraintes temporelles (systèmes d'exploitation, réseau, applications …).

De nombreuses études sont en cours pour tenter de définir une structure de système distribué temps réel (CORBA et JAVA temps réel, mémoire virtuelle distribuée temps réel…).

Les systèmes distribués temps réel reposent souvent sur des structures spécifiques (systèmes d'exploitation, hôtes, voies de communications …).

 Dans la plupart des cas, c'est l'électronique qui est utilisée pour assurer les fonctions vitales du système (comme par exemple un système d'exploitation ayant des couches basses directement implémentées en logique programmée).

 Ces systèmes distribués prennent souvent la forme de systèmes « prédéfinis fermés » (nous verrons cette définition plus tard dans ce cours).

1.2.2 Les avantages d'un système distribué

1.2.2.1 Partage de ressources

Lorsque plusieurs sites sont interconnectés, les utilisateurs d'un site A peuvent utiliser les ressources d'un site B.

Par exemple, une imprimante connectée sur le réseau du site B peut être utilisée par les membres du site A. Dans le même temps, un utilisateur du site B peut accéder aux fichiers situés sur un hôte du site A.

Dans le monde de l'informatique industrielle, un hôte d'un site A peut accéder à une carte d'acquisition d'entrées / sorties située sur un site B.

Les systèmes distribués fournissent, en général, des mécanismes permettant de traiter les informations d'une base de données qu'elle soit locale ou distante.

1.2.2.2 Accélération des calculs

Lorsqu'un calcul peut être décomposé en un semble de « sous-calcul parallélisables », les systèmes distribués permettent la répartition de cette charge sur différents sites.

Si un site est en position de surcharge, certains calculs peuvent être déportés sur un site moins chargé.

La répartition complètement automatisée de la charge de calcul est rare dans les systèmes commercialisés actuellement.

Le mécanisme de répartition de charge automatisée se rencontre sous le terme « load balancing » et est souvent implémenté dans les systèmes tolérants aux fautes.

1.2.2.3 Fiabilité du système

Lorsqu'un des sites d'un système distribué tombe en panne, les autres doivent pouvoir continuer à fonctionner.

Si l'ensemble des sites participent tous au traitement d'un ou de plusieurs flux de données communs, la panne d'un calculateur, entraîne le dysfonctionnement de l'ensemble du système. Ce risque peut être limité en ayant une redondance suffisante tant au niveau matériel qu'au niveau des données traitées.

Dans le cas d'un système tolérant aux pannes, une défaillance doit être détectée et une action doit être entreprise pour avertir de cette défaillance. Le système distribué ne doit plus utiliser les ressources du site en panne. Si la fonction préalablement prise en charge par le site défaillant peut être sous-traitée à un autre, le système doit s'assurer que le transfert de traitements se fasse correctement. Dès que le site est de nouveau opérationnel, le système distribué doit pouvoir le réintégrer en douceur. Ce mode de fonctionnement pose de nombreux problèmes complexes dont certaines solutions seront vues dans ce cours.

1.2.2.4 Communication entre les systèmes

Lorsque plusieurs systèmes sont interconnectés par un réseau de communication global, les hôtes de plusieurs sites peuvent échanger des données.

Au niveau du réseau, les données sont transmises, entre les processus, par des flots de messages. Cela de la même façon que sur un système centralisé.

Etant donné cette possibilité de transfert de messages, l'ensemble des fonctionnalités, accessibles au niveau utilisateur local, peut être étendue de manière à englober le système réparti. (transfert de fichiers complets, navigation sur le WEB, appel de procédures distantes …)

La distance entre les sites peut être plus ou moins grande sans que cela influe sur la stabilité des données échangées.

La généralisation des systèmes distribués pousse les entreprise à répartir les traitements de leurs données internes. Cette opération de décentralisation se nomme le « downsizing ».

La communication interprocessus, au sein d'un système distribué, peut être faite soit en mode « communication synchrone » soit en mode « communication asynchrone ».

2. Eléments de conception d'un système distribué

Ce chapitre présente quelques éléments permettant de mieux appréhender le déploiement d'un système distribué.

On peut catégoriser les systèmes distribués en deux familles voisines l'une de l'autre. Les systèmes « prédéfinis fermés » et les systèmes « ouverts ».

2.1 Systèmes distribués « prédéfinis fermés »

Dans cette architecture, l'aspect distribué du système est borné et déployé suivant une étude préalable qui a pour but de définir quel est le niveau de distribution et d'hétérogénéité que l'on souhaite supporter.

En règle générale, l'ensemble des sites déployés concourent tous au traitement de mêmes flux de données (eux mêmes identifiés lors de la conception). Une ou plusieurs applications (ou domaine d'application) sont destinées à être prises en charge par ce type de système.

On rencontre ce type de système distribué dans le monde de l'informatique industrielle (contrôle de chaîne de fabrication, régulation de trafic, divers systèmes d'aide à la navigation …).

Les limites de charge, les supports physiques de transport des données, les langages de programmation ainsi que les processeurs supportés par de tels systèmes distribués sont **prédéfinis** lors des différentes phases de conception.

Dans ce cadre, il est tout naturel que les limites d'extension (« l'évolutivité ») du système soient connues dès le départ du projet. Ces systèmes sont paramétrables au moyen de fichiers de description de l'ensemble des sites interconnectés. Cette définition « statique » les rend moins ouverts vers de nouvelles technologies.

Ces systèmes utilisent régulièrement des systèmes d'exploitation distribués afin de confier un certain nombre de fonctionnalités de base directement au noyau.

Des protocoles de communication propriétaires sont souvent employés afin de répondre à des contraintes particulières exprimées dans un cahier des charges (bus de communication spécialisé, microcontrôleur dédié …).

2.2 Systèmes distribués « ouverts »

Dans cette architecture, l'aspect distribué du système n'est limitée que par les dimensions à un instant donné de l'ensemble des sites constitutifs du système distribué.

Ces systèmes utilisent le plus souvent des réseaux et protocoles de communication standardisés (ETHERNET et TCP/IP par exemple), des hôtes commercialisés à grande échelle et ont pour vocation de rendre le plus transparent possible l'hétérogénéité de l'ensemble des sites interconnectés.

Les utilisateurs de ces systèmes distribués peuvent être mobiles. Le système doit donc permettre la connexion, d'un utilisateur, à partir de n'importe quelle machine du réseau. L'environnement de l'utilisateur est donc « transporté » vers son lieu de connexion.

L'ajout de nouvelles ressources ne doit pas poser de problème particulier et ne pas nécessiter l'arrêt de l'ensemble des sites.

Ces systèmes se caractérisent par leur capacité à s'adapter en souplesse à une charge croissante.

 Le « hot plugging » et le « hot swaping » sont des techniques qui favorisent l'évolutivité et la maintenabilité de tels systèmes distribués. Ces technologies sont souvent prises en charge par du matériel dédié.

2.3 Au sujet de la tolérance de panne

Comme nous l'avons dit au début du cours, les systèmes distribués sont souvent associés à une réflexion sur la tolérance de panne.

Le terme « tolérance de panne » est très large. Il regroupe les pannes de communication, les bugs informatiques, les pannes de machines, les défaillances de périphériques de stockage ou tout affaiblissement d'un support de données.

Les degrés de tolérance varient suivant les contraintes que doit supporter le système à déployer.

Un système tolérant aux pannes doit pouvoir continuer à fonctionner, éventuellement sous une forme dégradée, même après l'apparition de pannes ou d'états indésirables.

La dégradation doit cependant rester proportionnelle aux dysfonctionnements qui l'ont provoqué.


Malheureusement, la tolérance aux pannes des systèmes distribués commerciaux restent encore très limité (des outils sont fournis mais c'est souvent le programmeur qui doit implémenter la gestion proprement dite des pannes).

Quelques exemples :

- Le système « VAX Cluster » de la société DEC autorise plusieurs hôtes à partager la même grappe de disque. Lorsqu'un système tombe en panne, les utilisateurs peuvent continuer à accéder aux données situés sur la grappe à partir de n'importe quel autre système.

- L'UNIX de la société SUN (SOLARIS) est livré en standard avec un outil logiciel de gestion de disques durs dans une structure RAID 5.

- Le système QNX (système d'exploitation temps réel et distribué) autorise la répartition de charge sur deux connexions réseau par machine. Si l'une des voies de communication tombe en panne, l'autre connexion est alors utilisée. Le basculement est pris en charge directement par le micro noyau.

Une panne peut être déclarée alors qu'une ressource est utilisée de façon trop importante. Le temps d'accès devient trop long et le système la considère en panne.

La tolérance de panne et l'adaptabilité (« scalability ») d'un système distribué dépendent l'une de l'autre.

La tolérance de panne demande une conception rigoureuse. Un système distribué complètement tolérant aux pannes est une vue théorique !

 La tolérance aux pannes à un impact sur les coûts et délais de développement.

Tout service ayant une charge proportionnelle aux nombres de machines présente dans le système limitera la capacité d'évolution de l'ensemble.

La conception d'un système adaptable et tolérant aux pannes ne doit comporter aucun point de contrôle et aucune ressource centralisés.

Les serveurs de noms, les serveurs de fichiers et de bases de données centraux sont autant d'exemples de centralisation de l'information. Cette centralisation impose une asymétrie fonctionnelle entre les différents nœuds d'un système distribué.

L'idéal serait un système distribué parfaitement symétrique au niveau de chaque machine le composant. C'est à dire que chaque nœud au sein du système joue un rôle équivalent dans le fonctionnement d'ensemble.

Dans la pratique, une distribution complète, évolutive et tolérante aux pannes est impossible à obtenir. Par exemple, l'ajout de machines « diskless » rompt cette symétrie.

 La technique de « clustering » (fonctionnement en grappe), est une solution permettant de satisfaire, en partie, les contraintes énoncées. Dans cette structure, un serveur de grappe est utilisé pour servir le maximum de requêtes émanant des hôtes dont il est responsable. La répartition des hôtes sous les serveurs de grappe doit être étudiée avec beaucoup de soin de telle sorte que l'entité « grappe » puisse constituer l'élément de base pour l'accroissement de la taille du système distribué.

La redondance de tout ou partie des logiciels et matériels permet de maintenir les services du système sans dégradation visible (si le système sait procéder à une permutation de ressource « à chaud »).

2.3.1 La détection de panne dans un système distribué

Dans un système faiblement couplé (donc sans mémoire physique partagée), il est impossible de différencier une panne de liaison, de site ou la perte d'un message.

 On se contente de détecter l'un des trois phénomènes mais on ne sait pas les distinguer.

Sur détection d'un état de défaillance, une action doit être entreprise. La nature de cette action dépend de l'application en cours d'exécution dans le système distribué.

La détection de ces cas de dysfonctionnement peut se faire au moyen d'un protocole de « handshaking » :

Prenons, comme exemple, un système comportant deux sites : A et B. Ces deux sites sont physiquement reliés. A intervalle régulier (la fréquence dépend des contraintes de temps de réaction que l'on souhaite obtenir et du média de communication utilisé), chaque site envoie à l'autre un message « je fonctionne ».

Si le site A ne reçoit pas le message de B (dans un temps maximum déterminé) il peut supposer que la liaison est rompue, que le site B est en panne ou que le message de B est perdu. A peut réagir de deux façons a cet événement :

- il attend le message de B pendant une période supplémentaire.
- il envoie un message « fonctionnes tu ? » au site B par une autre route que la précédente (s'il en existe une).

Si le site A ne reçoit toujours pas de réponse de la part de B, il peut en déduire les hypothèses suivantes :

- le site B est « tombé ».
- la liaison directe, et celle de remplacement, entre A et B sont interrompues.
- les messages se sont perdus.

Malgré tout, le site A n'a aucun moyen de préciser quelle est la nature réelle de la panne.

2.3.2 Réaction face à une détection de panne

Supposons qu'un site « S » ait détecté une défaillance d'un site « D ». Il doit entreprendre des actions pour informer les autres sites du système distribué de la présence de cette panne.

- Il s'assure qu'il peut communiquer avec les autres sites. Si aucun site ne lui répond, il se déclare en panne et compte sur le fait qu'un autre système va détecter sa propre défaillance.

- En cas de défaillance de la liaison directe entre S et D mais maintient de la route de remplacement, le site S doit informer l'ensemble des autres sites afin que ceux-ci mettent à jour leurs tables de routage des messages.

- Si le site supposé en panne joue le rôle de coordinateur pour une fonction distribuée (par exemple la gestion des interblocages), une « élection » doit être entreprise de façon a transférer cette fonction sur un site en état de fonctionner. Dans le cas ou le coordinateur supposé en panne est en fait encore en activité (mais déconnecté d'une branche du système), le système distribué peut se retrouver dans une situation conflictuelle. Le fonctionnement de l'application peut être aléatoire si les coordinateurs concurrents sont responsables de la gestion des sections critiques d'accès à une ressource partagée.

2.3.3 La redondance pour aider à la tolérance de panne

La redondance matérielle et logicielle permet de maintenir la qualité de service d'un système distribué en cas de défaillance.


La redondance n'a d'utilité que si le système distribué sait détecter les pannes et prendre la bonne décision pour contourner la défaillance.

On distingue trois catégories de redondance :

- **La redondance « froide »** : dans cette configuration, la permutation de l'entité en panne se fait par suppression du système défaillant dans un premier temps. Après réparation, l'entité est réinsérée manuellement, en douceur, dans le système. Pendant le temps de réparation, l'ensemble distribué se trouve dans un mode dégradé.

- **La redondance « chaude semi active »** : dans cette configuration, une panne sur une entité partagée est détectée et le système distribué procède au basculement automatique sur l'entité de secours. Après réparation, l'entité est réinsérée manuellement, en douceur, dans le système. Pendant le temps de réparation, l'ensemble distribué présente des performances similaires à l'état précédent la détection de la panne.

- **La redondance « chaude active »** : dans cette configuration, une panne sur une entité partagée est détectée et le système distribué procède au basculement automatique sur une entité de secours. Après réparation, l'entité est réinsérée automatiquement dans le système. Les performances de l'ensemble distribué sont identiques : avant la détection de défaillance, pendant la phase de réparation et après la réapparition de l'entité partagée.

2.4 Structure logicielle des serveurs

La structure d'un agent responsable d'un service fourni par un serveur, est très importante. Elle agit sur la performance globale lors des périodes de pointe.


Une **« période de pointe »** est une période pendant laquelle le nombre d'utilisateurs accédant à un service augmente considérablement. Cet instant est aussi appelé **« pic de charge »**.

Prenons comme exemple un système de serveur de fichier. Lorsque le processus responsable de ce service reçoit une requête d'entrée/sortie sur son système de fichier local, il se trouve placé dans la file d'attente des processus en accès au disque dur local jusqu'à ce que l'accès physique au bloc de données à lire ou a écrire soit exécuté par le matériel. Pendant ce temps, le service est donc « suspendu ».

2.4.1 Structure multiprocessus

Il apparaît clairement qu'il faut affecter un processus particulier à chaque requête d'E/S provenant d'un client.

Dans ce cas, nous pouvons élire un agent qui s'occupe de réceptionner les requêtes et affecte le travail demandé à un processus dédié. Le schéma suivant montre la structure d'un tel système.

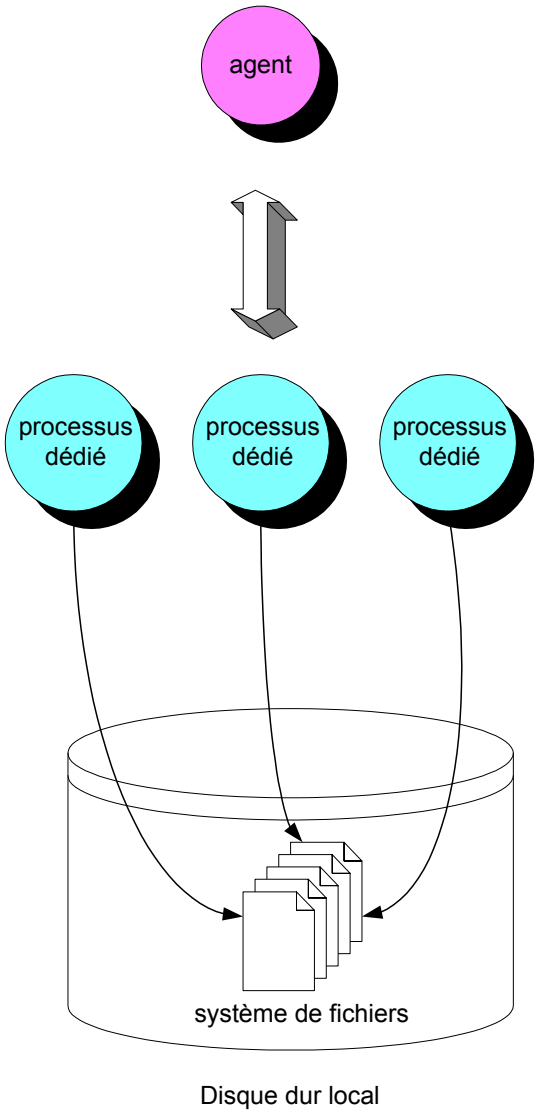


Figure 2.4.1-1 : service avec agent de contrôle

2.4.2 Structure multithreads

Avec une structure utilisant des processus de poids lourd, lorsque le nombre de processus dédiés augmente (ou lorsque le nombre de services offerts par le serveur s'accroît) les temps de commutation de tâches deviennent un « goulot d'étranglement » du système local.


La meilleur solution est donc l'utilisation de processus de poids léger comme les threads. Ceux ci présentent l'avantage de réduire considérablement le temps de commutation de contexte des processus.

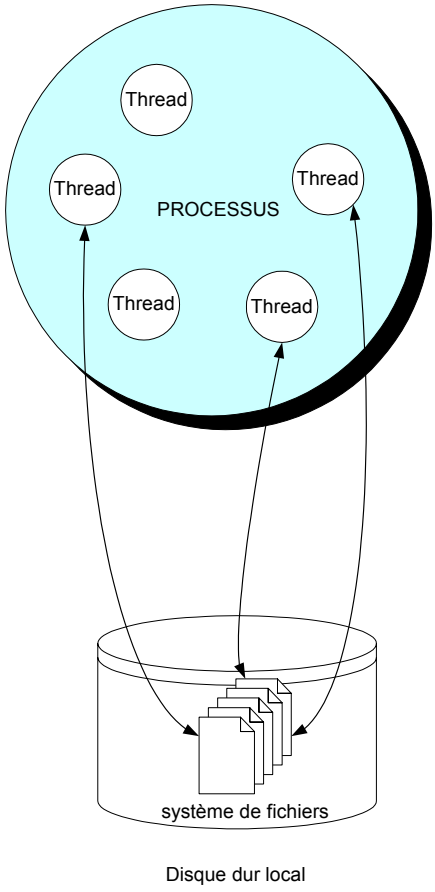


Figure 2.4.2-1 service multithreads

2.4.2.1 Rappels sur les threads

Le multithreading est une forme de multitraitement qui se repend de plus en plus. Il est donc important de bien connaître les différentes formes d'implémentation rencontrées dans les systèmes d'exploitation.

L'objet de ce cours n'est pas de décrire, dans le détail, les structures multithreads mais le choix de cette technologie requiert tout de même une connaissance minimum de ces entités.

La méthode de gestion des threads est importante pour choisir un système d'exploitation (ou une bibliothèque). Ce choix aura un impact non négligeable sur l'ensemble des traitements distribués (les temps de réponse).

Il m'a donc semblé nécessaire d'ajouter ce chapitre de rappels sur cette technologie dont on parle beaucoup mais qui n'est pas forcément toujours bien comprise.

 Il existe deux formes de thread :

- **Les threads noyau** : ces unités d'exécution sont directement supportées par le noyau du système d'exploitation (dans son espace d'adressage). Puisque c'est le noyau qui les gère, si un thread exécute un appel système bloquant, le noyau peut procéder à une commutation de thread et maintenir le processus qui les supporte sur le processeur (Windows NT, SOLARIS et DIGITAL UNIX supportent les threads noyau).

- **Les threads utilisateur** : ces unités d'exécution sont gérées (création, ordonnancement …) par des bibliothèques utilisateurs. Le noyau du système d'exploitation n'a donc aucune conscience de leur existence et si un thread exécute un appel système bloquant, tous les autres threads se trouveront bloqués à leur tour (Pthread POSIX et les threads de SunOS sont des bibliothèques).

 L'ordonnancement des threads peut être préemptif ou non. Dans le cas ou les threads sont ordonnés de façon non préemptive, leurs données partagées n'ont pas besoin d'être explicitement protégées. Dans le cas contraire, il est obligatoire d'utiliser un mécanisme de d'exclusion mutuelle entre les threads (en général un verrou, un mutex, un moniteur, …).

Attention : suivant le système d'exploitation, le support des threads peut-être différent. En effet, de nombreux systèmes supportent à la fois les threads utilisateurs et les threads noyau.

On rencontre trois modèles de multithreading différents :

- modèle « n vers 1 »
- modèle « 1 vers 1 »
- modèle « n vers n »

2.4.2.1.1 Modèle « n vers 1 »

Dans ce modèle, la gestion des threads est entièrement faite dans l'espace de l'utilisateur local. Tous thread faisant un appel système bloquant empêchera l'exécution des autres thread. (SunOS 4 fonctionnait avec ce modèle).

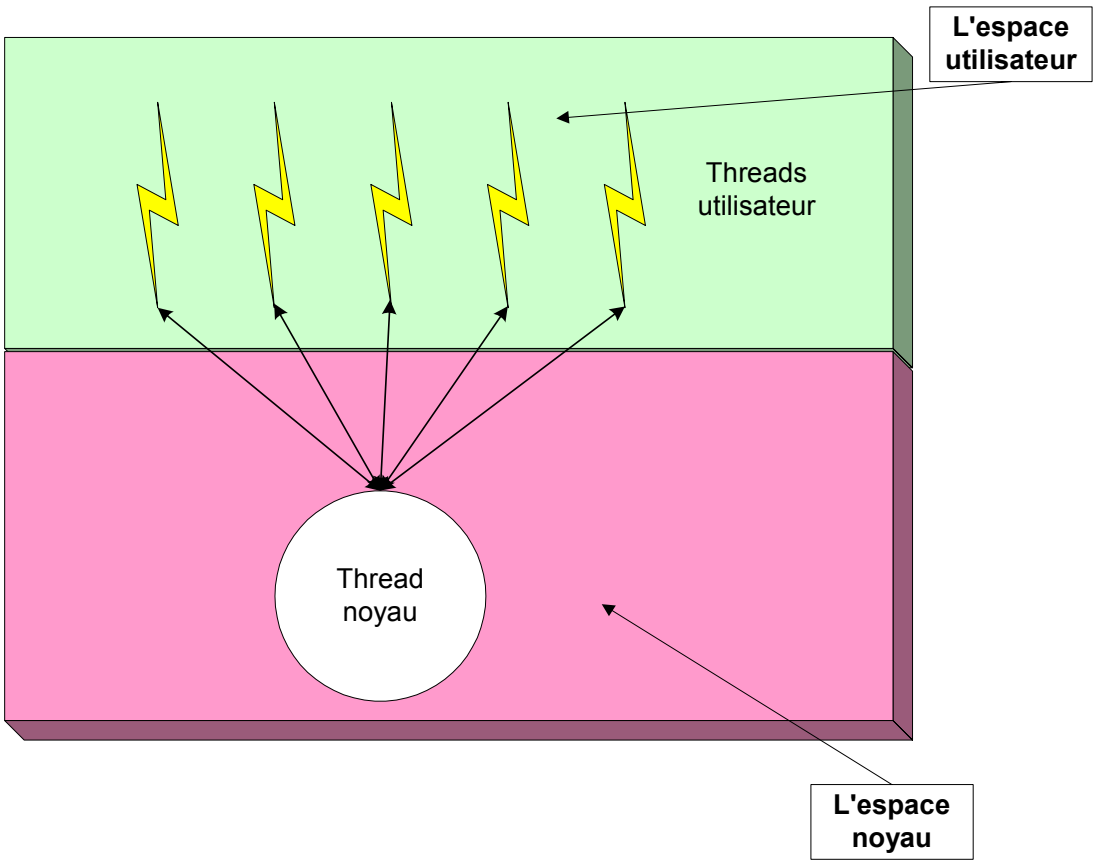


Figure 2.4.2.1-1 modèle "n vers 1"

2.4.2.1.2 Modèle « 1 vers 1 »

Dans ce modèle, chaque thread utilisateur est associé à un thread noyau. Un thread faisant un appel système bloquant n'empêche pas les autres threads de s'exécuter. Par contre, le nombre de threads utilisateur sera limité par le système d'exploitation lui-même. (Windows NT et OS/2 implémentent ce modèle).

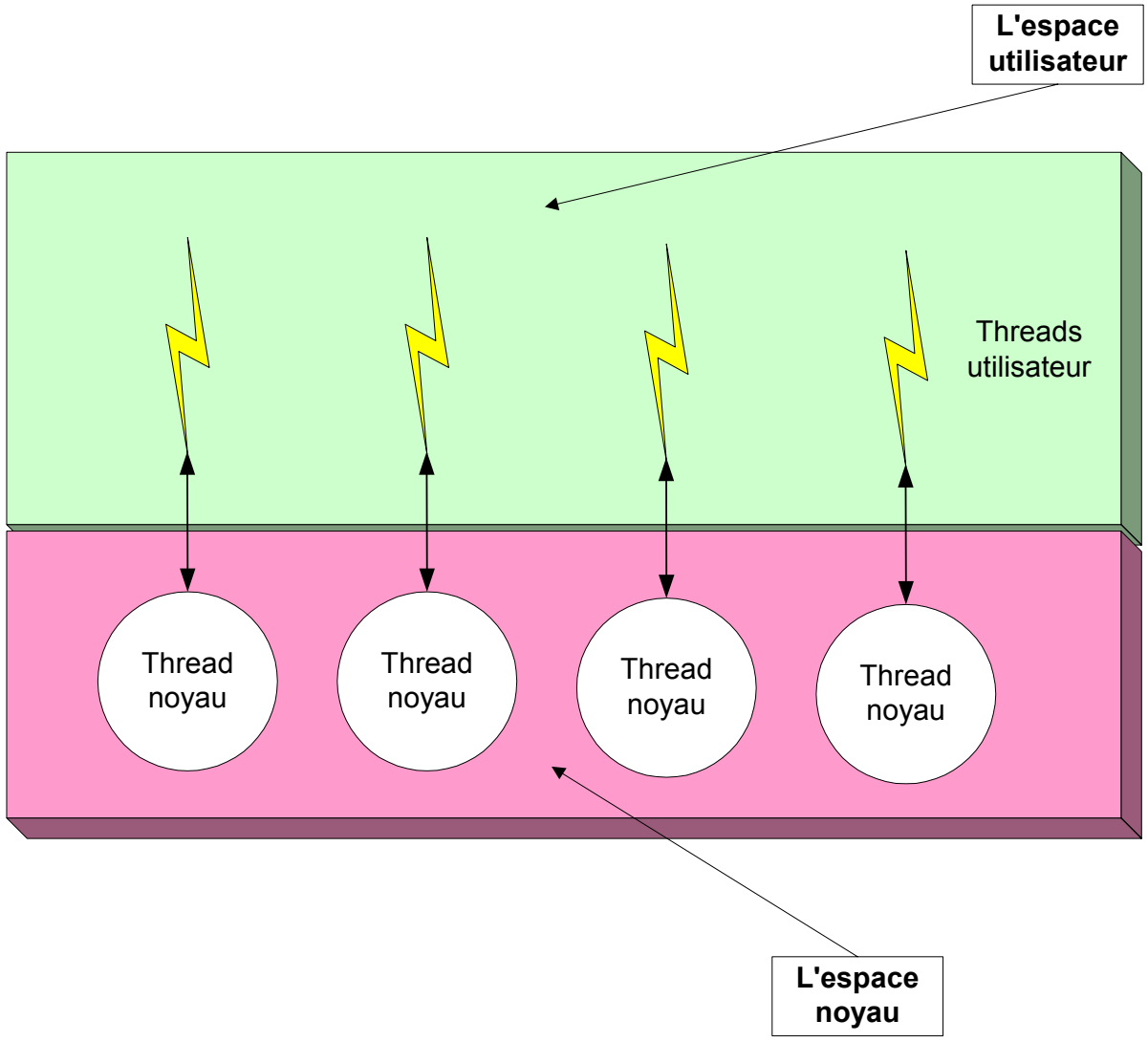


Figure 2.4.2.1-2 modèle "1 vers 1"

2.4.2.1.3 Modèle « n vers n »

Dans ce modèle, le nombre de threads utilisateur n'est pas limité par le système d'exploitation. Le nombre de threads noyau peut être égal ou supérieur au nombre de threads utilisateur. Sur un système multiprocesseurs, le noyau peut répartir les threads sur les différents processeurs, permettant de cette façon une réelle exécution concurrente des traitements.

Bien entendu, si un thread utilisateur exécute un appel système bloquant, les autres threads continueront de s'exécuter. (SOLARIS, Lynx OS et l'UNIX de DIGITAL supportent ce modèle).

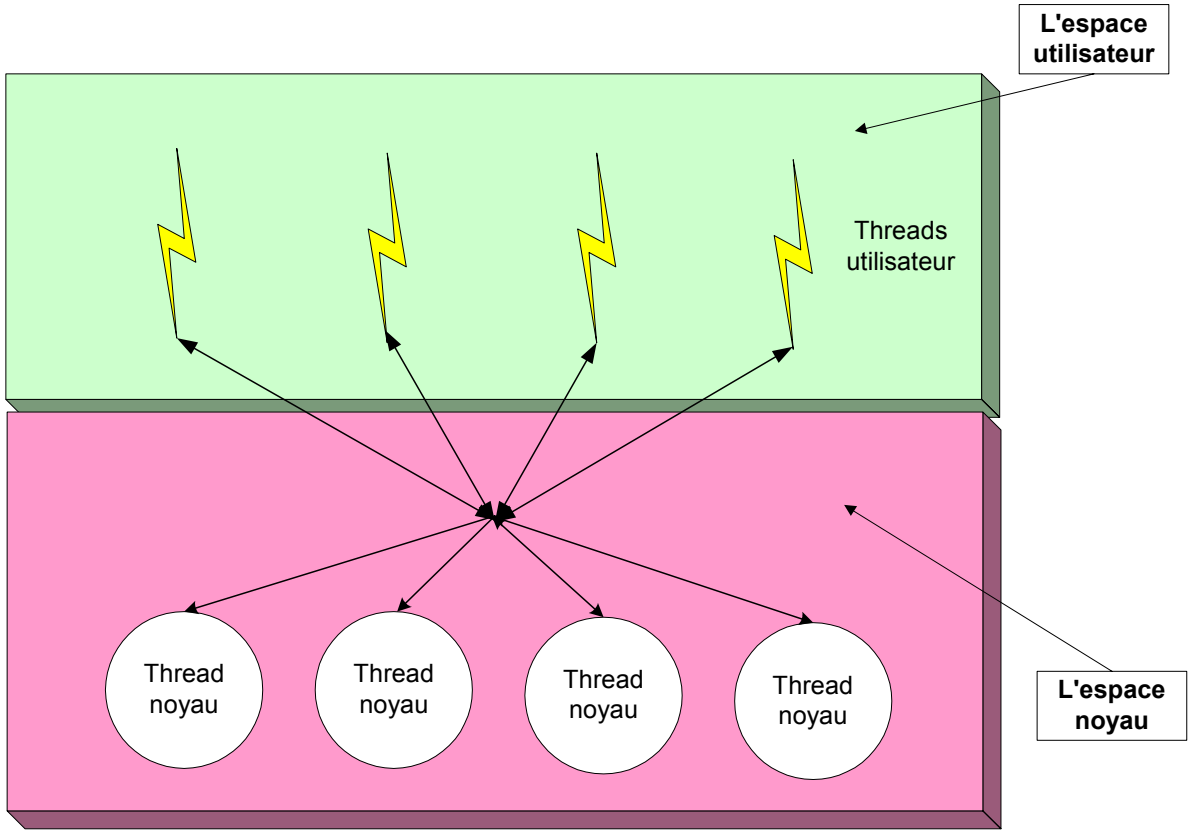


Figure 2.4.2.1-3 modèle "n vers n"

3. Types de systèmes distribués

Il existe trois grandes familles de systèmes distribués :

- Les systèmes distribués architecturés autour de systèmes d'exploitation orientés réseaux (OS supportant des protocoles tels que « Telnet » et « FTP »)

- Les systèmes distribués reposants sur des systèmes d'exploitation auxquels sont ajoutées des fonctionnalités « réparties » (systèmes supportant : des protocoles de migration de charge de calcul tels que RPC, des protocoles de migration de processus tel que les « applets JAVA », des protocoles de migration de données tel que NFS …). Ces systèmes permettent de gérer, de façon plus ou moins évidente, l'hétérogénéité des différents hôtes.

- Les systèmes distribués reposant sur un système d'exploitation complètement distribué (souvent rencontré sous la forme de micronoyaux). Dans ce cas, le noyau lui-même possède tous les outils pour prendre sous sa responsabilité la communication inter processus synchrone (fonctionnement par envoi de message). Ces systèmes ne gèrent pas directement l'hétérogénéité des différents hôtes.

 Nous ne traiterons, dans ce cours, que des structures distribuées reposants sur des systèmes d'exploitation auxquels sont ajoutées des fonctionnalités « réparties ». Cette structure est la plus couramment rencontrée.

4. Types de distribution

Dans un système distribué , la migration des données, des calculs ainsi que des processus entre deux sites est sous le contrôle de chaque système d'exploitation local.

4.1 Distribution de données situées en mémoire secondaire

Prenons comme exemple un utilisateur d'un site A souhaitant accéder à des fichiers d'un site B. Deux méthodes sont alors envisageables :

- Transfert complet (copie) de B vers A du fichier souhaité. Transfert inverse dans le cas ou l'utilisateur du site A a effectué des modifications sur le fichier. Cette technique n'est pas satisfaisante pour des modifications de petites tailles sur des fichiers volumineux. Le réseau supportera une charge induite importante dans le cas d'accès répétés de ce type.

- Transfert partiel (copie) en local de la zone à lire ou écrire. Dans ce cas, la bande passante du réseau n'est pas inutilement utilisée pour des accès répétés sur des zones réduites de fichiers volumineux. Par contre, dans le cas d'accès séquentiels sur des fichiers de taille importante, cette technique ne s'avère pas idéale. Le système de fichier distribué NFS (Network File system) de la société SUN repose sur le principe de transfert partiel (avec gestion de zones de cache).


Une contrainte importante dans les systèmes distribués est l'hétérogénéité des différents hôtes. En effet, les processeurs utilisés ainsi que les divers langages de programmation font que la représentation mémoire des valeurs manipulées varient d'un système à l'autre (« little endian » et « big endian », problème d'alignement de structures typées …).

Du fait de cette constatation, le système distribué doit fournir tous les outils nécessaires à la conversion des différentes grandeurs manipulées.

4.2 Distribution de calculs

Il est parfois plus performant de distribuer des calculs plutôt que les données associées. Cette technique est appelée « migration de calculs ».

Prenons, par exemple, une tâche qui a besoin d'accéder à plusieurs gros fichiers répartis sur le réseau. Il est plus optimum de procéder aux accès aux fichiers directement sur le site respectifs plutôt que de les transférer sur la machine ou s'exécute cette tâche.

 Lorsque le temps de transfert des données est supérieur au temps de traitement en local, cette méthode doit être utilisée.

La migration de calcul peut se faire en invoquant une RPC (« Remote Procedure Call ») dans les systèmes supportant ce type d'exécution distante. Les RPC utilisent un protocole de « datagrammes » (UDP pour la stack TCP/IP).

Attention : l'utilisation de la couche RPC nécessite obligatoirement, en environnement hétérogène, l'utilisation du niveau XDR (« External Data Representation »).

Avec le langage JAVA, vous pouvez utiliser les RMI (« Remote Method Invocation »).

CORBA permet aussi de faire migrer des objets sur un système réparti.

Si le système ne supporte pas l'invocation de procédures distantes, il faut mettre en place un système de déport d'exécution par envoi de message de commande. Un processus P1 (situé sur un site S1) désire effectuer un calcul sur le site S2. P1 envoi un message de demande d'exécution au site S2. Celui-ci crée un processus P2 dédié au calcul demandé. Lorsque P2 a terminé son exécution, il retourne le résultat au processus P1. Bien entendu, ce mécanisme est plus consommateur de ressource processeur que la méthode avec RPC.

La distribution de calcul peut très bien être cascadée (demande d'exécutions séquentielles) ou parallélisée (demande d'exécutions distantes sur plusieurs hôtes en même temps).

4.3 Distribution de processus

La distribution de processus est une extension du principe de distribution de calcul vu précédemment.

Lorsqu'un traitement est démarré sur une machine du réseau, il n'est pas obligatoire que le ou les processus associés soient exécutés en local.

Il peut être intéressant de faire migrer les processus eux mêmes (entièrement ou partiellement) pour les raisons suivantes :

- Equilibrage de charge : les processus sont répartis sur le réseau pour équilibrer la charge de travail
- Accélération des calculs : les calculs étant pris en charge par plusieurs processeurs en parallèle, le temps d'exécution global s'en trouve réduit.
- Dépendance d'un processus vis à vis d'un matériel : certains processus sont exécuter plus efficacement sur des processeurs spécifiques (comme par exemple un traitement d'une image sur un processeur graphique).
- Dépendance d'un processus vis à vis d'une autre application : le processus peut être amené à dialoguer avec une autre application qu'il est impossible de déplacer (taille, clés d'utilisation …)
- Accès à d'importants volumes de données : le processus est déplacé directement sur le ou les hôtes qui abritent les données à manipuler.

Le WEB est un exemple de système distribué qui utilise l'ensemble des mécanismes de distribution que l'on vient d'aborder. La migration de données s'utilise entre les serveurs et les clients. La migration de calcul se retrouve lorsqu'un client effectue des requêtes d'accès à une base de données distante. Et enfin, la migration de processus se retrouve lors de l'utilisation « d'applets JAVA ».

5. Implémentation des systèmes distribués

5.1 Les réseaux dans les systèmes distribués

Un système distribué s'architecture autour d'un ou plusieurs réseaux de communication entre les différents hôtes.

Ces réseaux se rencontrent sous deux formes :

- Les LAN (« Local Area Network »). Ce sont des réseaux qui couvrent une petite zone géographique (un ensemble de bureau, de bâtiment par exemple).

- Les WAN (« Wide Area Network »). Ce sont des réseaux qui couvrent de grandes distances (connexion entre deux villes, entre deux continents). Les liaisons utilisées dans ce type de réseaux ont tendance à être plus rapides et présenter un taux d'erreur de transmission supérieur au réseaux de type LAN.

 La qualité de service des réseaux dans les systèmes distribués doit être surveillée avec beaucoup de soin car ils représentent la colonne vertébrale de tous système de distribution.

5.1.1 Stratégie de « nommage » et résolution de noms

Sur tous les réseaux informatique, les différents systèmes sont nommés (de façon unique) de telle sorte que l'être humain puisse manipuler plus facilement les différentes entités interconnectées.


Dans tous les systèmes multitâches, les processus sont numérotés. Les différents hôtes ne partagent pas de mémoire, et ne peuvent donc pas connaître la localisation précise d'un processus distant. L'association du nom de la machine et de l'identifiant du processus permet de retrouver le processus sur le système distribué.


Puisque les machines sont désignés par des chaînes de caractères et que l'informatique préfère les chiffres (pour des raisons de simplicité et de rapidité), il faut donc qu'une traduction soit faite. On appelle cette traduction **« résolution de noms »**. Sous UNIX, cette traduction est faite au moyen du fichier « /etc/host » dans lequel on trouve une correspondance entre l'adresse IP et le nom de chaque machine.

Dans les systèmes distribués de grande envergure, il est difficile d'imaginer que chaque machine possède un fichier de traduction de l'ensemble des noms des autres systèmes. On utilise, en général, des mécanismes de résolution de noms en arborescence. Ces mécanismes font intervenir des serveurs de noms sous lesquels on trouve des grappes de machines. Ces serveurs sont les seuls à connaître l'ensemble des noms de machine qui leurs sont connectées. Un système couramment utilisé est le **DNS** (« Domain Name Service »).

5.1.2 Stratégies de routage

Lorsque deux systèmes ne sont connectés que par une seule voie de communication, l'ensemble des messages sont véhiculés via cette route.

Si les sites possèdent plusieurs voies de communication, le système doit choisir une voie par laquelle il fera transiter les messages. L'algorithme de choix de la route a emprunter est appelé **« routage »**.

Le routage peut être de différentes formes :

- **Routage statique** : dans ce cas, les chemins possibles sont définis à l'avance dans des tables de routage statiques. On choisit, en général, les chemins les plus court pour des raisons de performances ou de coût de communication

- **Routage virtuel** : dans ce cas, une route est déterminée au moment de l'ouverture d'une session entre deux machines.

- **Routage dynamique** : la route à utiliser est calculée dynamiquement pour chaque message envoyé d'un site vers un autre site. En règle générale, les « routeurs » dynamiques utilisent les liaisons les moins chargées de façon à rendre optimum la fluidité du trafic sur le réseau.

Le routage statique et le routage virtuel permettent d'assurer que les messages arriveront dans le même ordre que leur génération.

 **Le routage statique ne permet pas de tolérance de pannes**.

 **Le routage virtuel permet une tolérance de pannes limitée**. En effet, la route empruntée par les messages d'une session reste la même jusqu'à la fermeture de la session. Lors d'une nouvelle session, une nouvelle route est déterminée.

 **Le routage dynamique permet la tolérance de pannes même au sein d'une session déjà ouverte mais ne garantit pas que les messages arrivent tous dans le même ordre que leur génération**.

Un **« routeur »** est une entité particulière du système distribué. Il a en charge la détermination de la route a emprunter. Les « routeurs » utilisent des algorithmes particuliers pour effectuer un apprentissage des routes éligibles (« spanning tree » par exemple).

5.1.3 Stratégies de paquets

Les applications distribuées s'échangent des messages via le réseau. La longueur de ces messages est, en général, de taille variable.

Pour des raisons de simplification des couches basses, les réseaux transmettent des messages de taille fixe baptisés **« paquets »**, **« trames »** ou **« datagrammes »**. La transmission des paquets peut être faite en mode « fiable » ou non.

Dans le cas d'une transmission « fiable », l 'émetteur du message est assuré que le destinataire l'a bien reçu. En fait, le récepteur retourne un accusé réception à l'émetteur. Cet accusé réception peut être le message lui même ou un message spécifique. Bien entendu, lorsque l'émetteur ne reçoit pas l'accusé réception, il ne peut pas en déduire que le message n'a pas été reçu par le destinataire. Effectivement, l'accusé réception lui même peut avoir été perdu dans le réseau.

Si un message est trop long pour être **« encapsulé »** dans un paquet, le système établi une **connexion** entre les deux machines et transmet plusieurs paquets.

5.1.4 Stratégies de connexion

Dans une application multiprocessus distribuées, des connexions sont souvent établies directement de processus à processus. Ces connexions sont appelées des **« sessions de communication »**.

Les méthodes de communication directe entre des processus sont nombreuses. On peut en citer trois qui sont couramment utilisées :

- **Commutation de circuits** : dans ce cas, une liaison « physique » entre les processus est établie pour toute la durée de la session de communication. Ce type de connexion est dédié à une paire de processus de façon exclusive.

- **Commutation de messages** : dans ce cas, une connexion est établie entre les deux processus pour chaque transfert de message. Ce type de connexion permet à plusieurs paires de processus d'utiliser la même voie de communication.

- **Commutation de paquets** : si la taille d'un message complet dépasse la taille d'un paquet élémentaire, le système le décompose en plusieurs paquets qui ne vont pas emprunter nécessairement la route.

5.1.5 Gestion des conflits d'accès au réseau

Suivant la topologie du réseau du système distribué, il se peut qu'une liaison physique relie plus de deux ordinateurs (par exemple réseaux en anneau ou à bus à accès multiples).

Dans cette configuration, des trames peuvent être émise au même instant et aboutir à un mélange des informations qu'elles transportent.

Ce mélange d'information nécessite une relance de l'émission de la part de tous les systèmes ayant envoyé des trames dans le même créneau temporel.

Ce mélange de trame s'appelle une **collision**.

Plusieurs techniques sont utilisées pour détecter les collisions et apportent des solutions pour éviter l'écroulement de la bande passante du réseau :

- **CSMA/CD** (« Carrier Sense with Multiple Acces / Collision Detection ») : avant de transmettre une trame, un site vérifie qu'aucune donnée n'est déjà en cours de transport sur le réseau. Cela signifie que chaque site est en écoute permanente du réseau. Si la liaison est utilisée, le site doit attendre un silence de transmission pour entreprendre son émission. Dans le cas ou plusieurs sites détectent un silence en même temps, ils détectent une collision et interrompent leur transmission. A partir de cet instant, il doivent attendre pendant un temps aléatoire mais borné. Après expiration de ce délais d'attente, il recommence la transmission de leur trame. Le réseau Ethernet 802.3 repose sur ce principe.

- **Circulation d'un jeton** : un message particulier appelé **« jeton »** circule en permanence sur le réseau (généralement structuré en anneau). Lorsqu'un site désire émettre une trame, il retire le jeton, émet sa trame puis réinjecte le jeton dans le réseau. Si le jeton est perdu, une élection est faite pour déterminer quel site va générer un nouveau jeton. Le réseau Token Ring repose sur ce principe.

- **Créneaux de messages** : plusieurs créneaux de messages circulent en permanence sur le réseau (en général structuré en anneaux). Un site qui désire émettre une trame attend l'arrivée d'un créneaux vide dans lequel il insère les données à transmettre. Lorsque le site destinataire réceptionne le message, il déclare le créneau libre et le réinjecte dans le réseau.

5.2 Protocole de communication sur les réseaux

Nous avons vu qu'un système distribué repose forcément sur un réseau assurant l'interconnexion entre les différents sites qui le constitue.

Pour que la communication puisse s'établir, il est essentiel que l'ensemble des machines s'accordent sur un ou plusieurs protocoles de communication. On utilise en général une conception en couche pour chacun des protocoles supportés.

L'**ISO** (« Internationnal Standard Organization ») a défini un modèle à 7 couches. Ce modèle porte le nom de **« modèle OSI »** (« Open System Interconnexion »). Voici sa représentation :

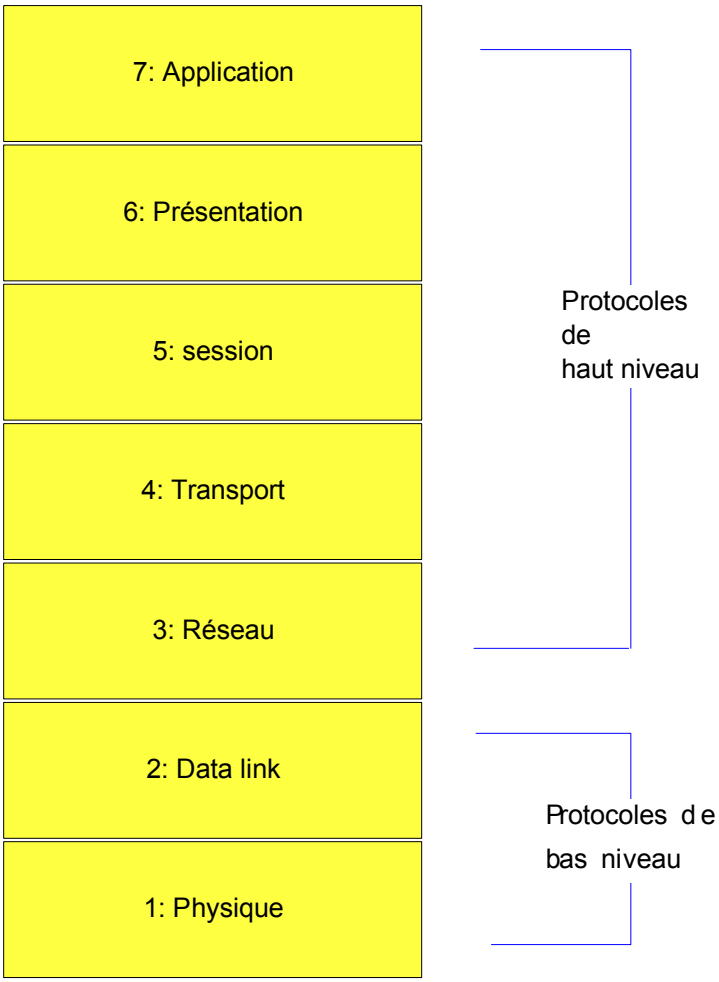


Figure 5.2-1 Le modèle OSI

Nous pouvons transposer la **« stack TCP/IP »** dans le modèle ISO. Dans les faits, ce protocole possède moins de couches que le modèle ISO puisqu'il remplit plusieurs fonctions dans chacune des couches (pour des raisons d'efficacité).

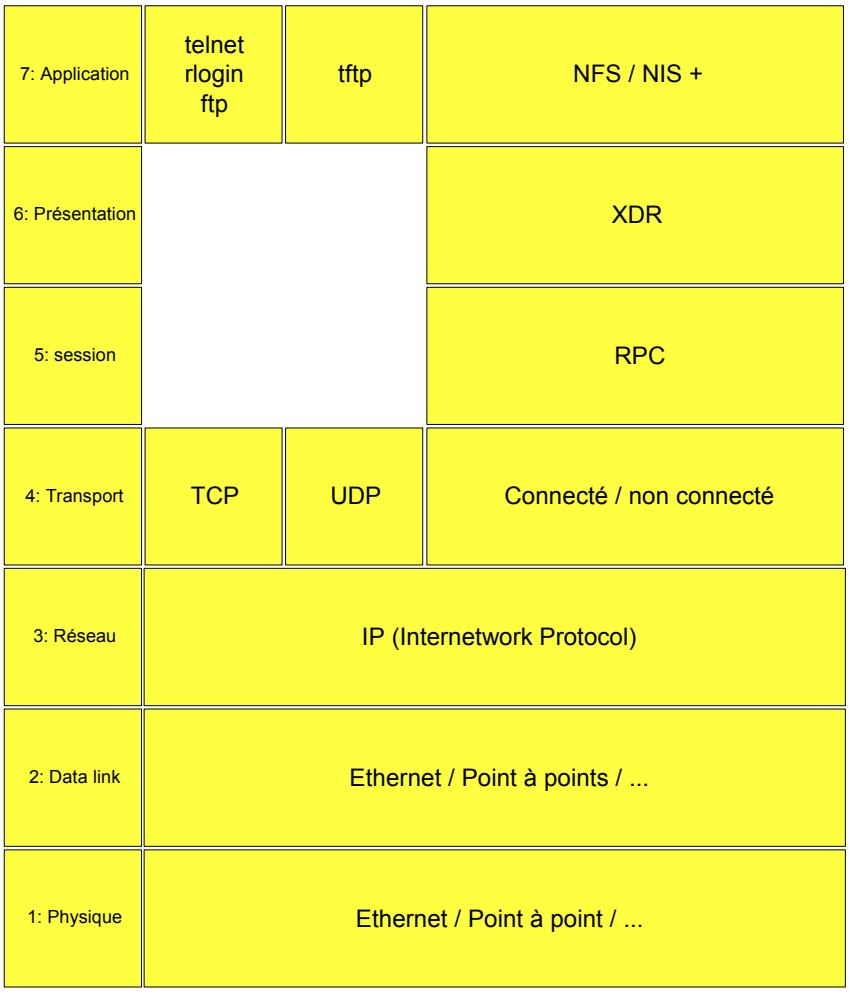


Figure 5.2-2 TCP/IP et le modèle OSI

5.3 La communication distribuée

Maintenant que nous connaissons les différentes structures des systèmes distribués, nous abordons les différents mécanismes de communication réparties.


Les technologies les plus utilisées aujourd'hui sont les suivantes :

- **Les Sockets**
- **Les communications synchrones : RPC ou « message passing » des micronoyaux**
- **Les RMI de JAVA**
- **Le bus à objets distribués CORBA**

Les sockets et les RPC ne permettent pas de gérer directement les différences de modèle mémoire qui peuvent exister entre deux machines différentes.

Les RMI sont spécifiques au langage JAVA. Cela impose l'utilisation d'un seul langage sur l'ensemble des hôtes qui désirent utiliser ce procédé.

CORBA est indépendant du langage utilisé et des machines d'exécution. Il peut donc faire cohabiter des systèmes parfaitement hétérogènes.

Le « message passing » impose d'implanter le même système d'exploitation sur les machines qui utilisent cette technologie.

5.3.1 Les sockets

Une socket est définie comme l'extrémité d'une voie de communication dans une paire de processus. Ce lien inter processus comporte donc obligatoirement deux sockets.

 Leur fonctionnement est proche des « pipes » UNIX.

Une socket est construite par concaténation d'une adresse IP et d'un numéro de port.

Les sockets sont utilisées dans bon nombre d'applications structurées en « client / serveur ».

L'établissement d'une connexion entre deux processus se fait de la manière suivante :

- Le serveur crée une socket et se positionne en écoute de demande de connexion.
- Un client fait une demande de connexion (création d'une socket locale).
- Le serveur accepte la connexion.
- Le client et le serveur sont connecté par le tube ainsi crée et peuvent lire ou écrire dans leurs sockets respectives. A partir de cet instant, chaque processus (ou thread) peut être producteur ou consommateur de données.

Rappel :

> les sockets servent à transmettre des flots d'octets non structurés. C'est au programmeur de développer un système d'identification des trames de façon à savoir quel sont les types de données manipulées (structures « C » plus ou moins complexes, tableaux de caractères …).
>
> Dans le cas où deux machines communiquent mais possèdent des processeurs dont le modèle mémoire n'est pas le même, il faut prévoir de convertir les données.
>
> Une solution élégante est d'utiliser la couche XDR (« External Data Representation »). Cette technique permet de travailler en environnements hétérogènes mais n'est pas simple a mettre en œuvre.

Note : Les services comme Telnet, FTP, mail et http utilisent les sockets.

5.3.2 La communication synchrone

On entend par **« communication synchrone »** tout procédé qui permet d'exécuter une section de code distante de la même manière que si l'appel était local. L'exécution se fait de façon séquentielle vis à vis du programme.

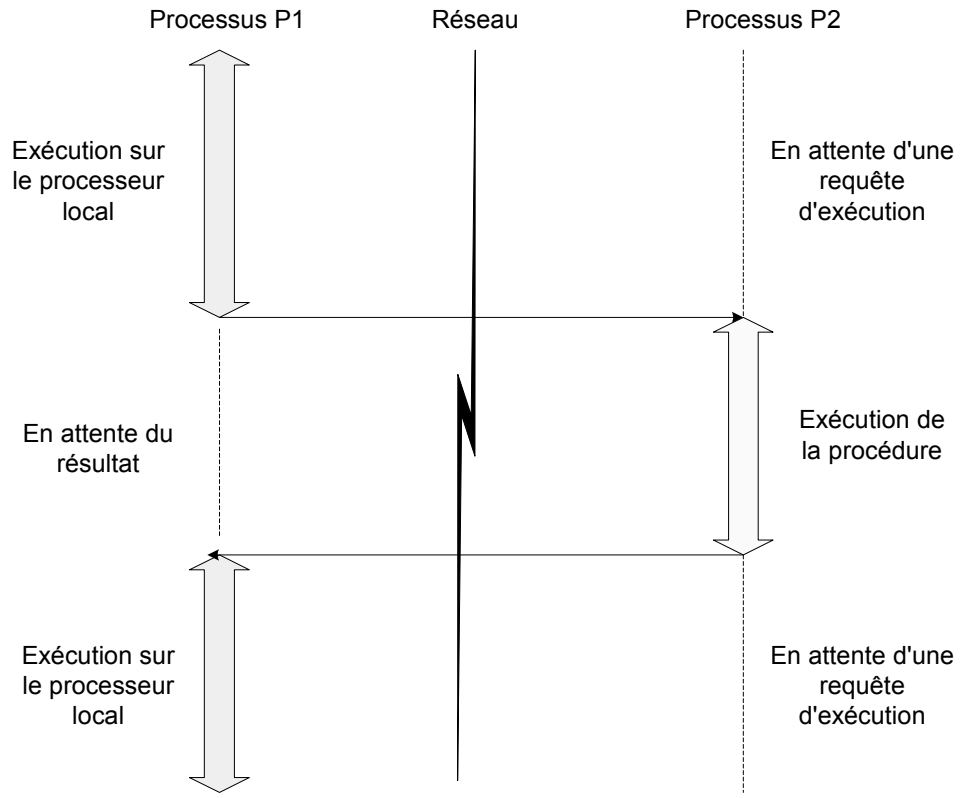


Figure 5.3.2-1 schéma d'une communication synchrone

Deux protocoles se rencontrent fréquemment dans les systèmes distribués. Les « RPC » d'une part et le « message passing » d'autre part.

5.3.2.1 Les RPC

Le système de socket, bien que très largement employé, est un système de bas niveau. Il ne permet que la distribution de données.

 Les RPC (« Remote Procedure Call ») ou appel de procédures distantes, permettent la distribution de calculs sur un système distribué.

Un processus (ou un thread) peut appeler une procédure située sur une autre machine que celle sur laquelle il s'exécute. Tout se passe, vis à vis de la conception d'un programme, de la même façon que si la procédure s'exécutait en local.

Ce sont les RPC qui gèrent le canal de communication avec le processus distant.

Le protocole RPC est indépendant d'une part des protocoles sous jacents utilisés pour l'échange de messages et d'autre part du système d'exploitation. Sous UNIX, le protocole sous-jacent est la plupart du temps UDP (plus rarement TCP).

Le système de fichier réparti NFS de la société SUN repose sur l'utilisation des RPC.

Le principe des RPC est de réunir un ensemble de procédures généralement relatives à un domaine d'application ou à un service particulier en un **programme RPC**.

Un programme sera identifié par un nombre entier et chaque procédure d'un programme y sera également identifiée par un entier. A titre d'exemple, le programme NFS porte le numéro 100003 et les procédures de lecture et d'écriture y possèdent respectivement les numéros 6 et 8. Enfin, pour permettre l'évolution des programmes, chaque programme possède un numéro de version.

 Dans la pratique, réaliser un appel RPC consistera à s'adresser à un processus démon de la machine où cette fonction doit être exécutée en lui transmettant en particulier :

- le numéro du programme
- le numéro de la version du programme à utiliser
- le numéro de la procédure à exécuter
- les différents paramètres de la fonction et leur type (les paramètres seront regroupés dans une structure à laquelle la fonction XDR correspondante sera appliquée)

Le processus démon se chargera d'établir un dialogue avec un processus qui exécutera la fonction demandée.

5.3.2.2 Le message passing dans les micronoyaux


Le message passing permet de créer un circuit virtuel entre deux processus. C'est une forme de communication synchrone entre deux processus.

Cette technique repose sur une structure client / serveur. Le serveur est en attente de réception de message. Le client initie un dialogue synchrone avec le serveur en lui envoyant des données.

Le client peut transférer des données vers le processus serveur et doit attendre la réponse de celui-ci pour continuer son exécution.

Le serveur peut retourner des données structurées comme réponse de la requête qu'il a traité.

Cette technologie est dépendante du système d'exploitation et impose donc que les machines souhaitant communiquer soient prises en charge par le même OS.

En règle général, le code source produit est strictement le même pour une communication locale ou pour une communication distante.

Un système de catalogue de noms, global à l'ensemble du système distribué, permet de créer les circuits virtuels entre les processus.

L'établissement d'un circuit virtuel, par le noyau du système distribué, se fait lors de la recherche d'un nom déterminé dans le catalogue global.

Le système d'exploitation distribué « QNX » utilise cette méthode de communication en natif pour toutes les communications entre le micro noyau et les différents agents du système.

5.3.3 Les RMI (JAVA)

Le langage JAVA augmente sa pénétration dans les systèmes informatiques distribués. Ce langage a été développé par la société SUN.

Les applications (ou « applets ») JAVA s'exécutent sur une machine virtuelle (JVM : « Java Virtual Machine »). Cette technique permet de rendre le code indépendant de la structure matérielle de la machine d'exécution.

JAVA est multithreads.

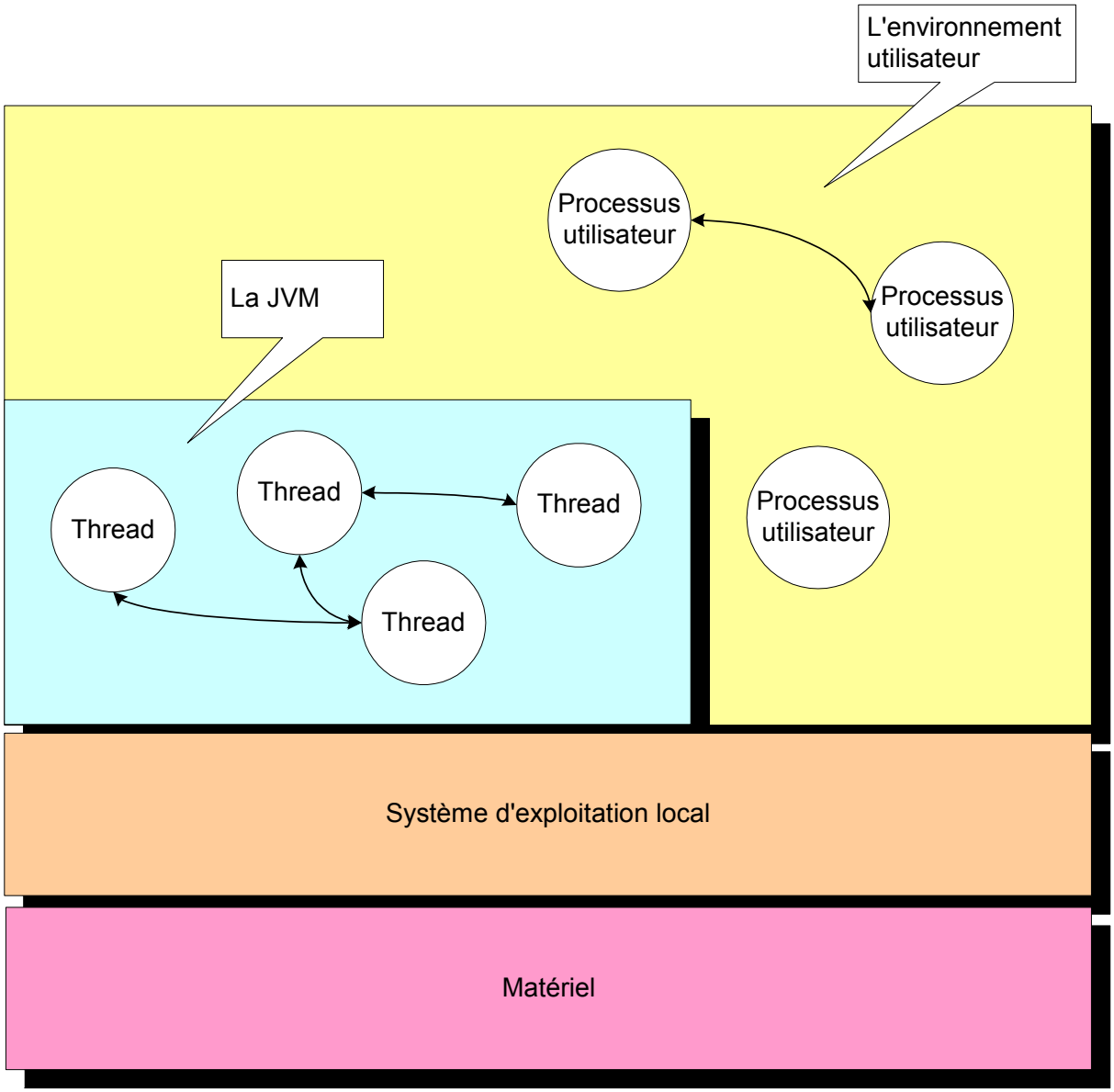


Figure 5.3.3-1 La JVM

Un thread peut invoquer une méthode appartenant à un objet distant. Un objet distant est un objet s'exécutant dans une autre JVM. Celle-ci peut très bien se trouver sur la même machine ou être située sur une machine distante.

RMI permet, contrairement aux RPC, d'invoquer une méthode distante en passant en paramètres des objets complets.

L'invocation de méthodes distantes passe par un système d'assemblage et désassemblage pris en charge par les « stubs » et les « squelettes ».

Le « stub » est implanté coté client et crée un paquet contenant le nom de la méthode à appeler ainsi que les paramètres associés. Cet opération est baptisée « assemblage ».

Le « squelette » reçoit les demandes venant des « stub ». Il est chargé du désassemblage des paquets et appel la méthode demandée par le client. Enfin il assemble la valeur de retour (ou éventuellement une exception) et la transmet au client.

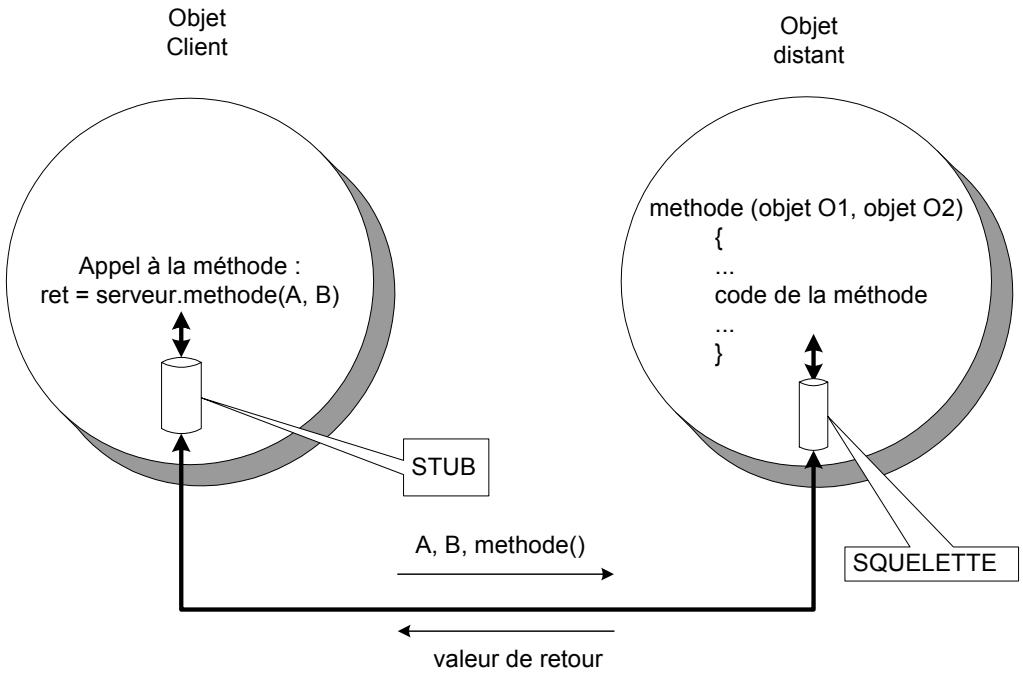


Figure 5.3.3-2 stub et squelette JAVA

Les stubs et les squelettes sont des entités complètement transparentes pour le programmeur JAVA.

Lorsque les paramètres sont des objets locaux, ils sont passés par copie. Cette opération se nomme **« sérialisation d'objet »**.

Si un objet distant est passé en paramètre, le stub client passera la référence sur cet objet. Cela permettra au serveur d'invoquer à son tour des méthodes de l'objet distant.

Pour qu'une méthode puisse être invoquée à distance, celle-ci doit être enregistrée dans un catalogue global nommé « RMI registry ».

La recherche d'un objet distant se fait par concaténation du nom IP de l'hôte et du nom de la méthode cherchée : « rmi://nom_hôte/nom_méthode ».


Cette méthode sous-entend que le programmeur connaît le ou les noms des machines qui exportent des objets.

Les méthodes inscrite dans le système RMI peuvent être déclarée « synchronized ». Cela permet de gérer simplement l'accès concurrent de plusieurs threads sur une même méthode.

Pour générer le « stub » et le « squelette », il faut utiliser le RMI compiler « RMIC ».


Les RMI sont une technologie JAVA vers JAVA. Cela signifie qu'une application distribuée reposant sur les RMI, est forcément développée avec le langage JAVA.

5.3.4 CORBA

CORBA est un « middleware » (couche logicielle intermédiaire) orienté objets.

CORBA signifie : « Common Object Request Broker Architecture ».

C'est l'OMG (« Object Management Group ») qui s'occupe de standardiser CORBA.

L'objectif de CORBA est l'intégration d'applications distribuées hétérogènes à partir de technologies objet indépendamment de :

- des moyens de communication réseaux
- des langages de programmation
- des systèmes d'exploitation

Toutes les requêtes à des objets distant passent par un ORB (« Object Request Broker »).

L'ORB d'un client doit trouver l'objet distant dans le système distribué. Il communique avec L'ORB serveur. L'ORB est appelé « bus logiciel ».

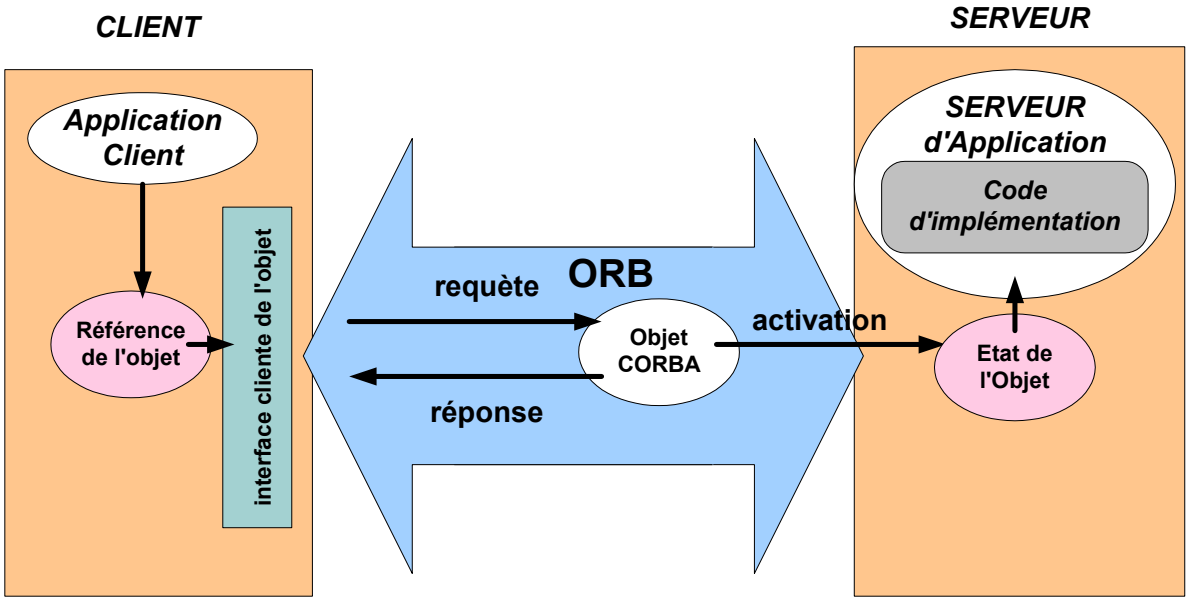


Figure 5.3.4-1 communication client/serveur avec CORBA

Comme pour les RMI, les objets s'inscrivent dans un catalogue global au système. Pour pouvoir communiquer avec un objet distant, un client doit au préalable trouver le nom de l'objet dans le catalogue.

Un système CORBA fonctionne de la façon suivante :

> Dès qu'un client a obtenu une référence à un objet distant, toutes les invocations aux méthodes de cet objet passent par l'ORB (via le stub client). Lorsque l'ORB du serveur reçoit une requête, il appelle le « squelette » approprié. Celui ci invoque à son tour la méthode demandée. Des valeurs de retour peuvent être renvoyée par ce même canal de communication.

La figure suivante représente l'architecture du bus CORBA, conforme à la définition de l'OMA (« Object Management Architecture ») définie par l'OMG.

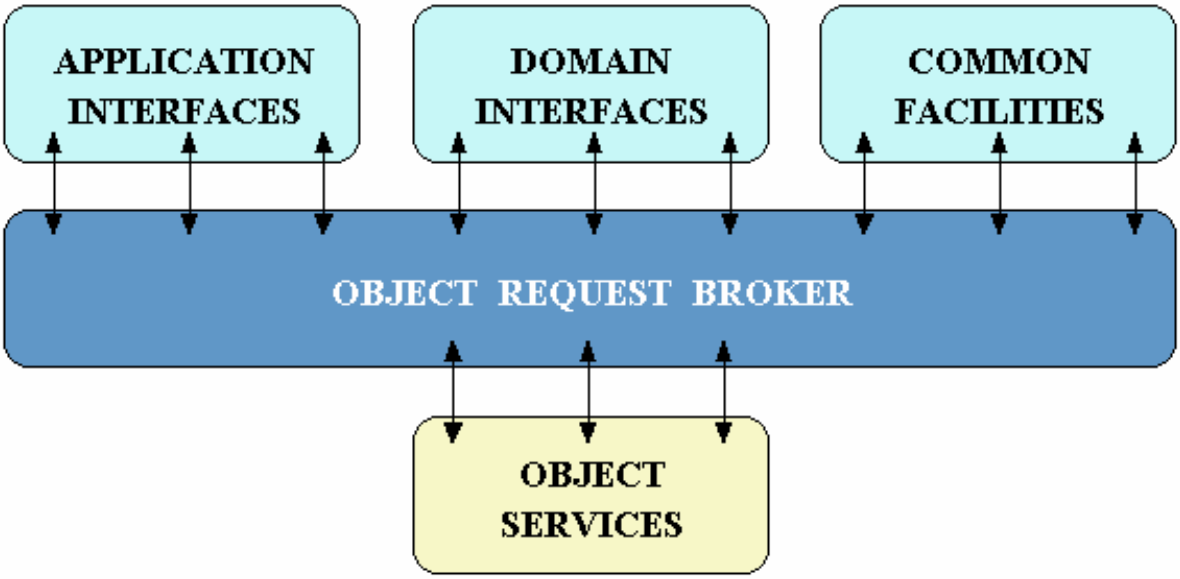


Figure 5.3.4-2 l'ORB

L'ORB (ou bus Bus d'Objets Répartis) est la clé de voûte du système. Il offre un environnement d'exécution aux objets distribués afin de masquer l'hétérogénéité des langages, des systèmes d'exploitation, des processeurs et des réseaux.

Le service d'objets commun fournit les services de base de gestion des objets (durée de vie, nommage, relations entre objets, transactions, sécurité …). L'OMG fait évoluer ces services en permanence.

Les utilitaires communs standardisent l'interface utilisateur, la gestion de l'information, l'administration …

Les interfaces de domaines fournissent des objets spécifiques à des métiers comme la finance, la santé, les télécoms… Leur objectif est de permettre une interopérabilité entre les systèmes d'informations d'entreprises évoluant dans un même métier.

Les interfaces d'applications contiennent tous les objets spécifiques aux différentes applications développées autour du bus CORBA.

5.4 La coordination répartie

Dans un système multitâches, l'ensemble des processus manipulent des données. Les différents traitements effectués sont souvent régulés par l'arrivée de messages (évènements de différentes formes).

Comme dans une application centralisée, les processus ont la nécessité de se synchroniser. Par exemple, l'accès à une ressource partagée nécessite des mécanicismes d'exclusion mutuelle globaux à une application (et donc distribués sur le réseau).


De plus, dans un système distribué, chaque machine possède une horloge locale. Cette horloge est souvent associée à l'heure humaine. Il est évident qu'en l'absence d'algorithme spécialisé, aucune machine ne sera positionnée exactement sur la même heure.

Il est donc nécessaire de posséder une horloge globale au système distribué.

Pour des raisons évidentes, la fiabilité d'un système distribué est une contrainte qui doit être présente pendant toute la durée de conception.

Les défaillances dans les canaux de communication entre les processus peuvent être à l'origine de défaillance générale du système.

5.4.1 Ordre des évènements / horloges

Dans une application distribuée, il est parfois nécessaire de posséder des mécanismes qui permettent de « dater » les différents évènements de notre système.

Nous distinguons deux types d'horloges :

- l'horloge physique
- l'horloge logique

5.4.1.1 Horloges physiques

Chaque système local possède une horloge physique (H_i) maintenue à jour par son système d'exploitation. Cette horloge est souvent basée sur le temps humain. Elle permet de dater les évènements gérés par le système. Ces horloges peuvent être plus ou moins précises et présenter des différences d'exactitude par rapport au temps « humain ».

Si il est nécessaire de se référer à une horloge globale en rapport avec le temps « réel » (humain), le système distribué doit alors maintenir une horloge « physique » globale à l'ensemble des machines. Les propriétés attendue d'une horloge globale sont les suivantes :

- Précision : $\forall T, | H_i(T) - H_j(T)| \leq \delta$
- Exactitude : $\forall T, | H_i(T) - T| \leq (T \pm \mu)$

Plusieurs algorithmes de synchronisation des horloges physiques sont rencontrés.


Dans tous les cas, la mise à jour d'une horloge locale ne doit se faire que par rattrapage positif de l'heure et JAMAIS en négatif. Imaginez un système d'archivage qui stocke des informations en fonction de l'heure locale. Si l'horloge est reculée, la cohérence des données n'est plus maintenue.

La fréquence de synchronisation des horloges doit être déterminée en fonction des contraintes que le système doit supporter.

Exemple d'algorithme de synchronisation d'horloges physiques :

Deux machines (M1 et M2) sont interconnectées par un réseau dont le temps de réponse est connu.

- M1 envoie à M2 son heure système à une date D1.
- M2 reçoit le message venant de M1 à une date D2 par rapport à l'heure de M1.
- M2 envoie un message vers M1 contenant son heure système : H(M2)
- M1 reçoit le message venant de M2 (contenant H(M2)) à une date D4
  - M1 calcule $\delta T = H(M2) - (D1 + T_{transmission})$: estimation de l'écart existant entre les deux horloges H(M1) et H(M2) à l'instant D1.
- Si M2 est en retard : M1 envoie un message à M2 indiquant le delta à rattraper
- Si M2 est en avance : M1 rattrape son retard.

5.4.1.2 Horloge logique

Il est souvent impératif de connaître l'ordre d'arrivée de deux évènements.

Dans une application centralisée, nous pouvons presque toujours déterminer à l'avance l'ordre d'exécution des traitements. Par contre, dans un système distribué, rien ne garantit que les messages échangés entre les machines arrivent dans le même ordre que leur génération.

La figure ci-dessous met en évidence les problèmes générés par le non respect systématique de l'ordre d'arrivée des messages dans un système distribué.

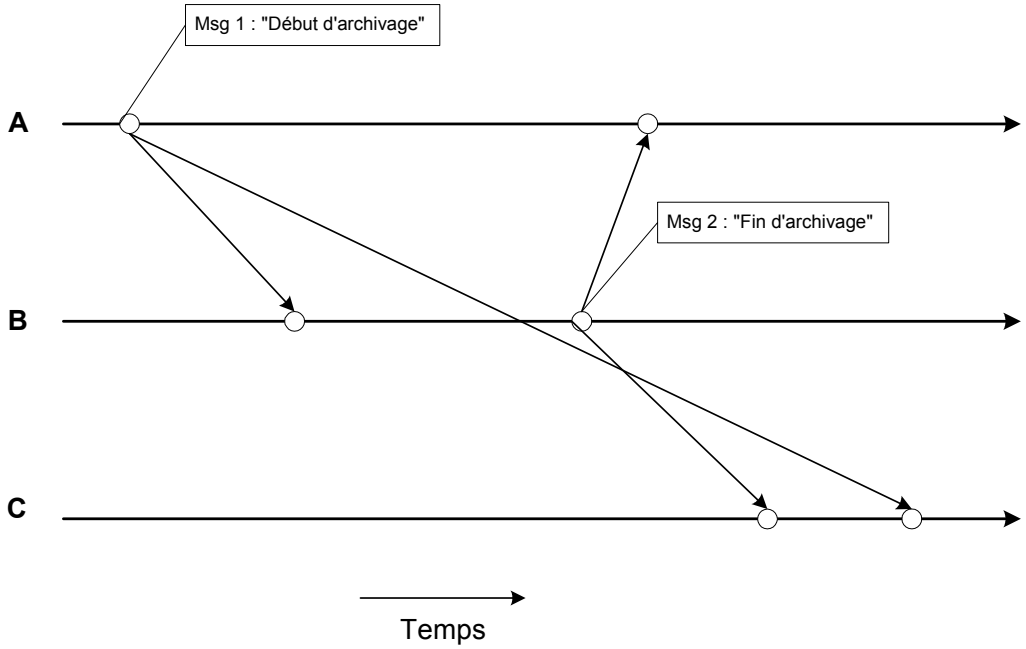


Figure 5.4.1.2-1 ordre des évènements

5.4.1.2.1 Relation de précédence et concurrence

Les processus qui s'exécutent sur un processeur sont considérés comme des séquences d 'exécution parfaitement séquentielles.

Il est évident qu'un message ne peut pas être reçu avant d'avoir été envoyer. Nous pouvons donc définir une relation de précédence entre les différents évènements gérés par un système distribué.

La relation de **précédence** (« happens before relation »), notée par le caractère « → » se définie de la façon suivante :

- si E1 et E2 sont deux évènements générés par le même processus d'une machine et si E1 est généré avant E2, alors on peut écrire « E1 → E2 ».
- Si l'événement E2 déclenche, dans un autre processus l'événement E3, alors on peut écrire : « E1 → E2 » et « E2 → E3 » donc « E1 → E3 ».
- Si les deux évènements Ei et Ej ne sont pas liés par une relation de précédence, on dit alors qu'ils sont **exécutés en concurrence** : on représente cette concurrence de la façon suivante : « Ei || Ej ». Il est impossible de savoir quel événement s'est produit avant l'autre. Lorsque Ei se produit, on ne sait pas si Ej s'est déjà produit.

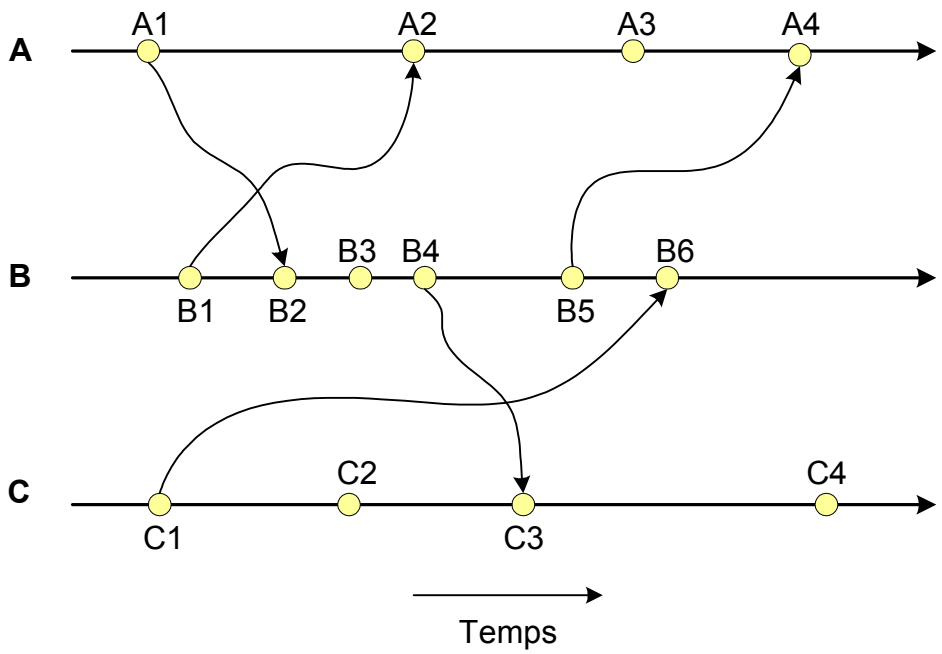


Figure 5.4.1.2-2 précédence et concurrence

Sur la figure précédente, on peut déduire les relations de précédence suivantes :

$A1 \rightarrow B2$

$B4 \rightarrow C3$

$C1 \rightarrow B6$

Mais aussi :

$A1 \rightarrow B4$ (car $A1 \rightarrow B2$ et $B2 \rightarrow B4$)

$B4 \rightarrow C4$ (car $B4 \rightarrow C3$ et $C3 \rightarrow C4$)

On peut aussi déduire, de la figure précédente, les évènements concurrents suivants :

$B3 \,||A3$

$C1 \,||\, A2$

$C2 \,||\, B3$

 Tous les processus désirant connaître l'ordre de deux évènements concurrents doivent s'accorder sur la façon de les ordonner.

5.4.1.2.2 Algorithme de l'estampille

Pour déterminer l'ordre de génération de deux évènements A et B, produits par deux systèmes différents, il est nécessaire de posséder une **horloge logique** commune à l'ensemble des machines du système distribué.

L'algorithme présenté ici est appelé **« algorithme de l'estampille »** ou **« algorithme de LAMPORT »**.

On associe une « estampille » (« timestamp ») à chaque événement du système. De cette association découle une exigence d'ordre globale : si « A $\rightarrow$ B », alors l'estampille de A est forcément inférieure à celle de B

Pour satisfaire à cette exigence, nous associons à chaque processus Pi, une horloge logique, Hli. Cette horloge peut être implémentée sous la forme d'un simple compteur que l'on incrémente entre deux évènements successifs générés par un processus. Elle permet d'affecter un numéro unique à chaque événement.

On en déduit donc :

Si « A $\rightarrow$ B », alors Hli(A) < Hli(B)

L'estampille d'un événement est en fait la valeur de son horloge logique.


L'algorithme mis en place fonctionne bien pour deux évènements générés par le même processus. Par contre, si on considère deux processus qui s'exécutent sur deux processeurs ayant des fréquences de travail différentes, l'exigence d'ordre global n'est plus garantie.

Pour illustrer cela, prenons deux processus P1 et P2. P1 s'exécute sur un système plus rapide que le processus P2.

Le processus P1 envoie un message à P2 : (événement E1). HL1(E1) = 500.

P2 reçoit le message (événement E2). L'horloge logique de P2 progresse moins vite puisque celui-ci s'exécute sur un système plus lent. Nous avons à cet instant : HL2(E2) = 350 (par exemple).

La situation obtenue transgresse donc notre exigence puisque :

E1 $\rightarrow$ E2 mais HL1(E1) > HL2(E2)

Pour résoudre ce problème, tous processus qui reçoit un message dont l'estampille est supérieure à la valeur courante de son horloge logique doit avancer celle-ci. L'horloge logique sera positionnée en ajoutant « + 1 » à la valeur de l'estampille reçue.

Pour illustrer cet algorithme, reprenons l'exemple précédent :

> Le processus P1 envoie un message à P2 : (événement E1). HL1(E1) = 500.
>
> P2 reçoit le message (événement E2). L'horloge logique de P2 progresse moins vite puisque celui-ci s'exécute sur un système plus lent. Nous avons à cet instant : HL2(E2) = 350. P2 détecte alors un problème dans la valeur des estampilles et positionne son horloge logique sur 501.

Voici une représentation de l'évolution des estampilles pour trois processus a, b et c :

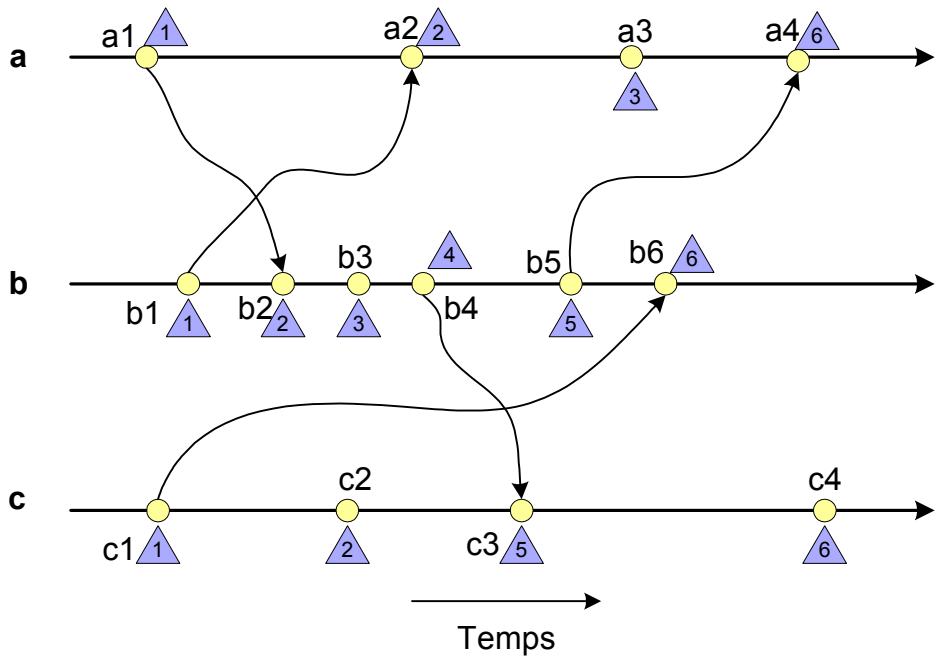


Figure 5.4.1.2-3 évolution des estampilles

5.4.2 L'exclusion mutuelle dans les systèmes distribués

Le partage de ressource demande souvent, de la même façon que dans un système centralisé, de gérer des sections critiques. Une section critique impose donc un mécanisme d'exclusion mutuelle entre les processus.

De cette façon, nous pouvons garantir qu'il n'y aura jamais plus d'un processus, à la fois, qui se situera dans sa section critique.


Afin de faciliter la compréhension des mécanismes étudiés dans la suite de ce cours, nous considérons deux conditions :

- Un processeur ne peut exécuter qu'un seul processus.
- Tous les processus du système distribué possèdent des numéros d'identification uniques.

Il existe au moins trois algorithmes de gestion de l'exclusion mutuelle. Un algorithme associé à une approche de **« décision centralisée »** et un algorithme associé à une approche de **« décision complètement répartie »** et un algorithme basé sur la « **circulation d'un jeton »** d'autorisation d'entrer dans une section critique.

5.4.2.1 Exclusion mutuelle : approche centralisée

L'approche centralisée doit son nom au fait que, dans cette structure, un processus unique dans le système distribué est choisi pour être le **coordinateur** (ou **agent**) de l'entrée dans la section critique concernée.

Un processus souhaitant pénétrer dans sa section critique, envoie une demande d'autorisation (« requête de réquisition ») au coordinateur. Dès qu'il reçoit un message de confirmation, il entre dans sa section critique.

Lorsque le processus sort de sa section critique, il envoie un message au coordinateur afin de lui signifier la **libération** de la ressource protégée.

Sur réception d'une requête de réquisition, le coordinateur exécute l'algorithme suivant :

> Si aucun processus est dans la section critique ➔ envoie du message d'autorisation
>
> Si un processus est déjà dans la section critique ➔ mise en file d'attente de la requête. Dès que le coordinateur reçoit un message de libération, il retire une des requêtes en attente et envoie un message d'autorisation au processus concerné. Notez que le choix de la requête à traiter en premier peu répondre à différents algorithmes d'ordonnancement (par exemple en FIFO).

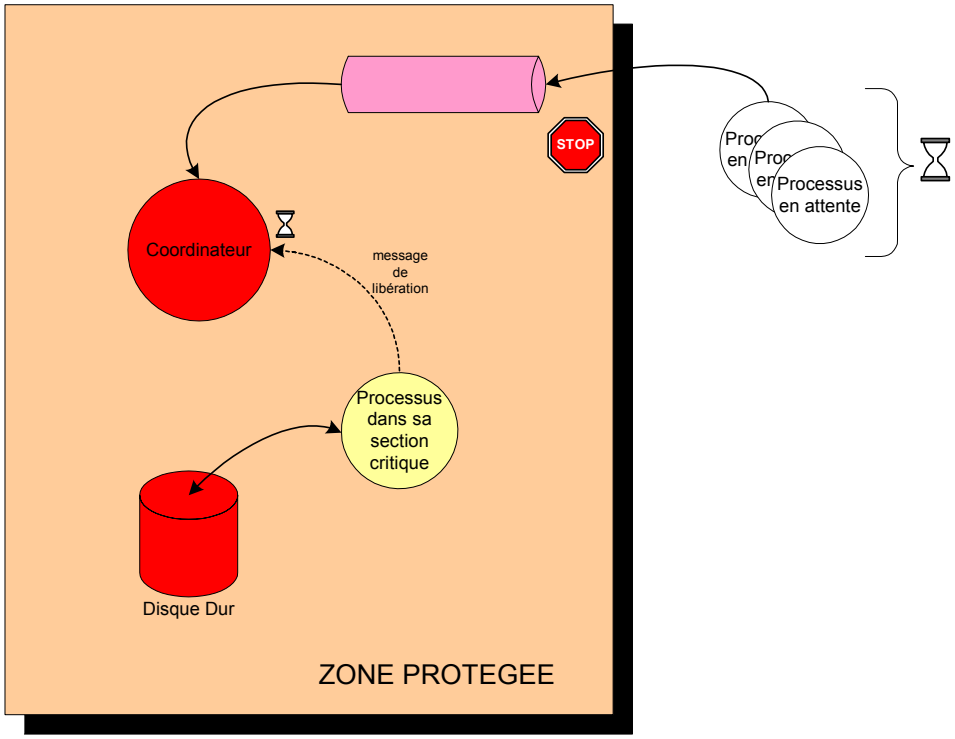


Figure 5.4.2.1-1 Coordination centralisée

Cette approche permet de gérer correctement les sections critiques. De plus, si l'ordonnancement choisi pour le coordinateur est équitable (par exemple la politique FIFO), aucun cas de **« famine »** ne peut se produire.

Cette méthode nécessite la génération de trois messages par entrée dans une section critique. Un message de demande, un message d'autorisation et un message de libération.

5.4.2.2 Exclusion mutuelle : approche distribuée

Il est tout a fait possible de répartir sur l'ensemble des processus, la prise de décision concernant l'entrée dans une section critique. Cette distribution de la prise de décision est plus compliquée à mettre en œuvre que l'approche centralisée.

Cette méthode génère un important trafic sur le réseau car elle nécessite l'envoi de nombreux messages entre tous les processus du système distribué. Elle ne s'applique donc qu'aux applications ne possédant qu'un nombre limité de processus concurrents.

Du fait des difficultés supplémentaires rencontrées en cas de défaillance d'un des processus (tous les autres doivent être informés de cette défaillance), l'approche répartie s'adapte mal à un système dont la stabilité n'est pas assurée.

Un processus P_1 désire entrer dans sa section critique. Il envoie un message à tous les autres processus (y compris à lui même). Ce message est constitué de son numéro de processus et d'une estampille T_{p1}.

Sur réception d'un message de demande de section critique, un processus à deux choix :

- **réponse immédiate** : cas ou le processus ne souhaite pas entrer dans sa section critique ou si il souhaite y entrer mais que son estampille est supérieure à $T_{p1.}$
- **réponse différée** : cas ou le processus est déjà dans sa section critique.

Le processus demandeur ne pourra pénétrer dans sa section critique que lorsqu'il aura reçu toutes les réponses des autres processus. Cela sous entend que ce processus à conscience du nombre de processus qui sont concurrents sur l'accès à la ressource protégée.

5.4.2.3 Exclusion mutuelle : méthode du jeton

Pour assurer une exclusion mutuelle distribuée avec la méthode du jeton, il faut au préalable que les processus soient organisés en anneau logique (il n'est pas nécessaire que le réseau lui-même soit un anneau).

Un jeton est un message particulier que les processus font circuler, en permanence, sur l'anneau logique.

Lorsqu'un processus désire entrer dans sa section critique, il doit attendre de posséder le jeton. A l'arrivée de celui-ci, il exécute les opérations suivantes :

1) réception du jeton.
2) passage dans la section critique.
3) sortie de la section critique.
4) remise en circulation du jeton dans l'anneau logique.

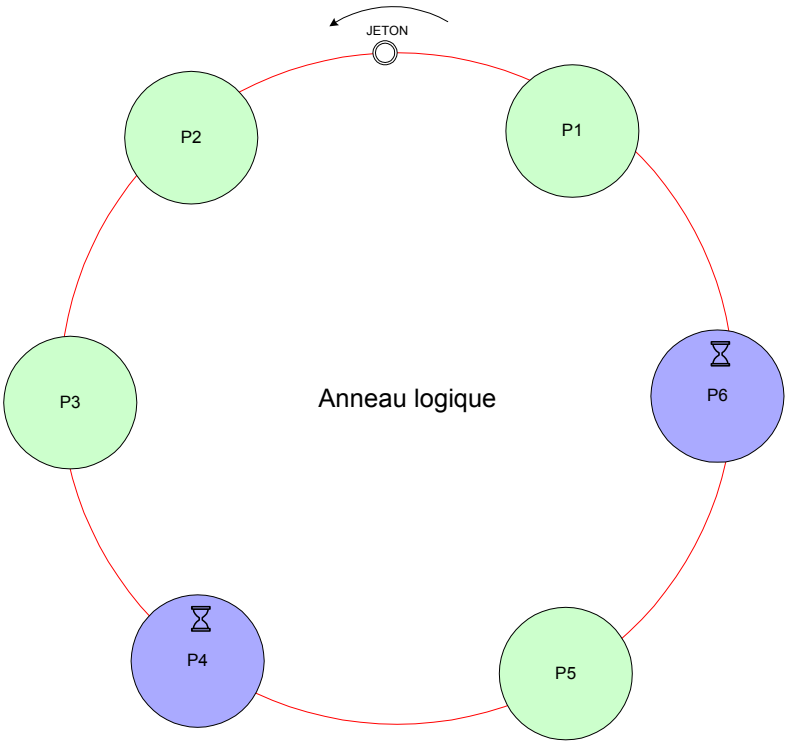


Figure 5.4.2.3-1 exclusion mutuelle par jeton

L'anneau logique peut être soit unidirectionnel soit bidirectionnel. Dans le cas d'un anneau unidirectionnel, il ne peut pas se produire de cas de famine.

Le temps d'attente moyen d'un processus pour entrer dans sa section critique dépend directement du nombre de processus constituant l'anneau logique.

Le temps d'attente moyen dépend aussi des caractéristiques du réseau sur lequel repose l'anneau logique.

En cas de perte du jeton, il faut procéder à une élection afin de déterminer quel processus de l'anneau va mettre en circulation un nouveau jeton.

En cas de défaillance d'un processus, l'anneau logique doit être reconstruit.

5.4.3 Transaction (atomicité)

Concept de haut niveau permettant de gérer l'exclusion mutuelle, la tolérance aux pannes et les inter blocages dans un système distribué.

Une transaction est une unité de travail élémentaire devant s'exécuter totalement ou pas du tout, et dont les effets ne doivent affecter les autres utilisateurs que quand elle est validée.

Une transaction peut être décomposée en plusieurs sous-transactions confiées à des sites différents.

Propriétés d'une transaction : « ACID »

- Atomicité (« All or nothing »)
- Cohérence (« consistency »)
- Isolation (pas d'interférence provenant d'autres transactions)
- Durabilité (« durability »)

Dans un système distribué, il est difficile d'assurer l'atomicité des transactions car plusieurs processus (ou sites) peuvent participer à la réalisation d'une transaction unique.

La défaillance d'un des participants à la transaction ou d'une des liaisons de communication peut aboutir à des résultats erronés.

La distribution de calcul impose à chaque site du système de posséder un **coordinateur de transactions** local.

Un coordinateur de transaction est responsable de la coordination et de l'exécution de toutes les transactions effectuées sur son site d'implantation :

- il démarre l'exécution de la transaction globale
- il découpe la transaction en une série de sous-transactions qu'il confit aux sites adéquats.
- Il coordonne la fin de la transaction. Cette terminaison peut aboutir sur la validation ou l'annulation de la transaction globale. Dans le cas d'une annulation, tous les sites ayant participer doivent aussi annuler leur transaction locale.

Pour satisfaire ces exigences, chaque site doit maintenir un journal des transactions pour assurer les cas de reprise ou annulation.

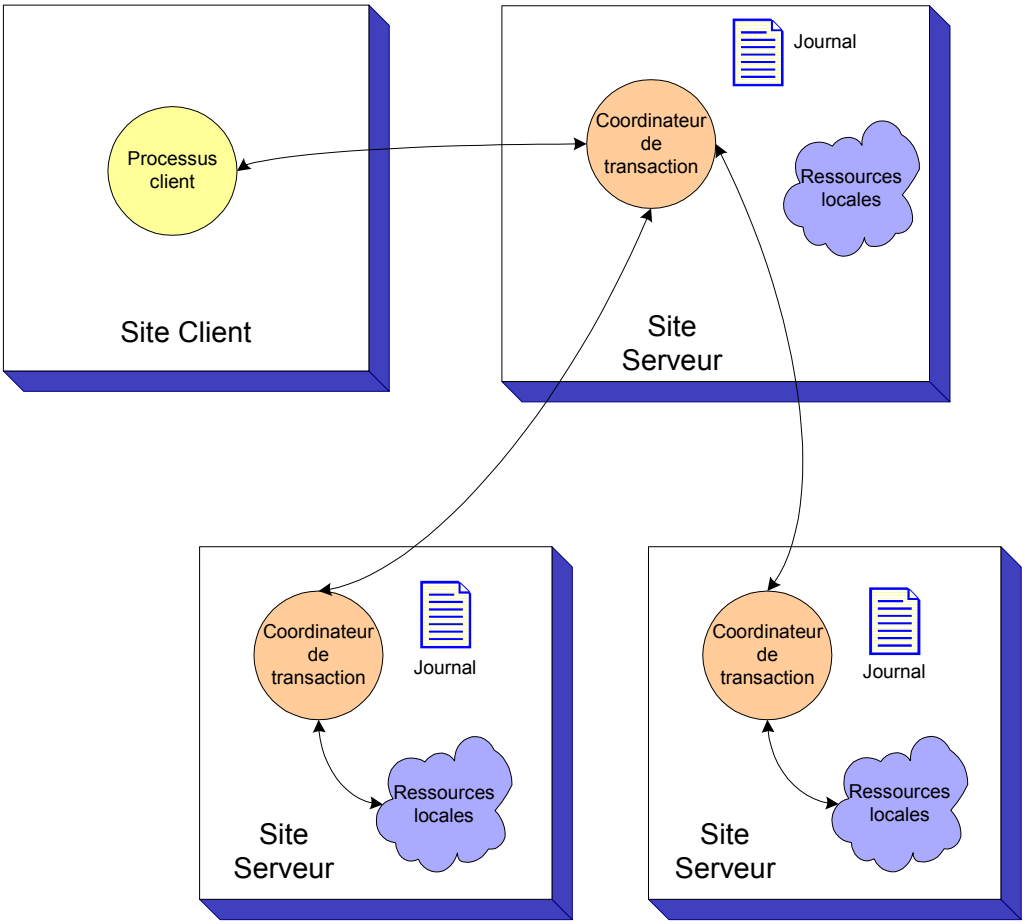


Figure 5.4.3-1 transaction atomique

5.4.3.1 Protocole de validation à deux phases : « 2PC »

Pour assurer que l'atomicité soit respecter, tous les sites participant à l'exécution d'une transaction « T » doivent se mettre d'accord sur le résultat de l'exécution de « T ».

Soit la transaction est validée, soit elle est annulée et cela sur l'ensemble des sites participants à l'exécution de « T ».

Pour coordonnée cette prise de décision, le coordinateur utilise un **protocole de validation**.

Le protocole le plus largement utilisé est le **protocole à deux phases** (« Two Phase Commit »).

Voici le fonctionnement du protocole 2PC. Nous partons d'un système dont un site « Sc » déclenche une transaction « Ti » prise en charge par un site « Si » dont le coordinateur de transactions est « Ci ».

Le coordinateur Ci répartit les sous-transactions « T » sur les différents sites concernés. Lorsque tous les sites ont terminés leur transaction « T », ils informent Ci de cette terminaison. Le coordinateur déclenche alors le protocole de validation à deux phases pour déterminer si la transaction doit être validée.

Ce protocole repose sur l'obtention de l'unanimité pour la prise de décision sur l'aboutissement d'une transaction.

Phase 1 : « obtenir une décision de tous les intervenants »

« Ci » ajoute l'enregistrement <prepare T> au journal. Il envoie ensuite à l'ensemble des sites participants, un message <prepare T>. A réception d'un tel message, le coordinateur local détermine si il désire valider la sous transaction qui lui a été confiée :

- Si sa réponse est non, il ajoute un enregistrement <no T> dans son journal et envoie <abort T> à « Ci ».
- Si la réponse est oui, il ajoute un enregistrement <ready T> a son journal et envoie un message <ready T> à « Ci ».

Phase 2 : « valider ou annuler »

Quand « Ci » reçoit toutes les réponses des sites concernés, il peut déterminer si la transaction T peut être validée ou annulée.

La transaction est validée si tous les sites ont répondus par <ready T>. Si un site n'a pas répondu (on peut borner l'attente par un time out) ou si une réponse <abort T> est parvenue, la transaction est annulée.

- si la décision est « valider T », le coordinateur de transaction Ci ajoute un enregistrement <commit T> dans le journal et envoie à tous les sites participants un message <commit T>. Sur réception de ce message, le coordinateur local d'un site valide la transaction et écrit dans le journal l'enregistrement <commit T> (On peut ajouter un message de confirmation de validation à destination de Ci).
- Si la décision est « annuler T », le coordinateur de transaction Ci ajoute un enregistrement <abort T> dans le journal et envoie un message <abort T> à l'ensemble des sites participants.


Nota : les journaux doivent être stockés dans une mémoire secondaire en plus de la mémoire principale. Cela permet la reprise d'une transaction dans le cas ou un site reprend après une défaillance.

5.4.4 Contrôle des accès concurrents

Il faut garantir la propriété d'isolation des transactions, lorsque dans un système distribué, plusieurs transactions sont exécutées simultanément par différents processus (sur différents processeurs).

Le gestionnaire de transactions d'un système de bases de données distribuées doit supporter les accès (transactions) concurrents aux données dont il a la responsabilité.

De plus, il doit aussi être capable de retrouver l'ordre des demandes puisque, comme nous l'avons déjà vu précédemment, l'ordre d'arrivée des demandes de transactions n'est pas garantit (voir chapitre sur la relation de précédence des évènements).

Nous allons donc nous intéresser d'une part aux algorithmes de gestion des verrous distribués et d'autre part à la gestion des relations d'ordre entre les transactions.

5.4.4.1 Protocoles de verrouillage

Il est donc nécessaire d'implémenter un mécanisme pour empêcher les transactions d'interférer. Ce mécanisme est géré par un algorithme de contrôle de concurrence ou de concomitance (« concurrency control »).

Plusieurs protocoles existent : « sans duplication de données », « coordinateur unique », « majorité », « estampillage » …

Les chapitres qui suivent abordent un certains nombre de protocoles de verrouillage rencontrés dans les systèmes distribués.

5.4.4.1.1 Verrouillage : politique « sans duplication de données »

Dans ce schéma, aucune donnée n'est dupliquée sur l'ensemble du système distribué.

Chaque site possède un **gestionnaire de verrouillage** permettant de gérer l'accès aux données locales.

Quand une transaction désire verrouiller une donnée sur le site Si, elle envoie une demande de verrouillage au gestionnaire du site Si. Le gestionnaire peut alors soit :

- accepter la demande et retourner une confirmation au demandeur
- différer la demande si la ressource est déjà verrouillée

Cette politique est simple a mettre en oeuvre Elle génère deux messages par demande de verrouillage et un message pour la demande de déverrouillage.

 Cependant, puisque chaque site s'occupe de gérer les verrous locaux, le traitement des interblocages est plus dur a mettre en œuvre.

5.4.4.1.2 Verrouillage : coordinateur centralisé

Dans cette structure, chaque site possède une duplication des données. Un processus (sur un site) est élu pour être le coordinateur central. C'est ce processus qui reçoit l'ensemble des requêtes de verrouillage et déverrouillage pour toutes les données distribuées.

Sur réception d'une demande de verrouillage, le coordinateur central peut réagir de deux façons différentes :

- il accepte la demande si la ressource n'est pas déjà verrouillée (ou si le mode de verrou demandé est compatible avec le verrou actuel : verrou partagé ou exclusif).

- Il diffère la réponse jusqu'à ce que la requête puisse être satisfaite.

Dans le cas ou la requête est acceptée, le site demandeur peut effectuer tous les accès qu'il souhaite sur tous les sites où sont dupliquer les données. Dans le cas d'une écriture, tous les sites possédant une duplication de la donnée doivent participer à l'écriture en local.

Cette politique centralisée présente deux avantages :

1) l'implémentation est simple

2) le traitement des interblocages utilisés pour un système centralisé peuvent être utilisés directement dans le gestionnaire de verrouillage (graphe d'allocation de ressources).

Par contre, cette politique présente deux problèmes majeurs :

1) le site qui abrite le gestionnaire de verrouillage centralisé devient un goulot d'étranglement

2) En cas de défaillance du site ou du processus de gestion des verrous centralisés, l'ensemble des requêtes sont abandonnées et un algorithme doit permettre la reprise après erreur (« journalisation » par exemple).

5.4.4.1.3 Verrouillage : décision « à la majorité »

Ce protocole est une extension de la politique « sans duplication de données ».

Il est destiné à un système qui autorise la duplication sur plusieurs sites mais avec un gestionnaire de verrouillage local.

Quand une transaction désire obtenir un verrouillage d'une donnée, elle doit envoyer une requête à plus de la moitié des sites possédant une copie locale. Chaque gestionnaire local décide si la demande peut être accordée ou non. Dans le cas ou la ressource est déjà verrouillée, le gestionnaire de verrouillage diffère la réponse au demandeur.

La transaction n'utilise pas la donnée tant qu'elle n'a pas obtenue la majorité.

Ce protocole présente l'avantage de traiter le verrouillage et la duplication de données de façon distribuée mais présente deux inconvénients :

1) l'implémentation de ce protocole est plus compliqué que les protocoles vu précédemment. Il requiert 2 * (n/2 + 1) messages pour gérer les requêtes de verrouillage et (n/2 + 1) messages pour traiter les requêtes de déverrouillage.

2) Des interblocages peuvent survenir. Imaginez un système composé de quatre sites qui désirent verrouillés de façon exclusive une donnée élémentaire dupliquée sur les chacun de ces sites. Supposons que la transaction T1 réussisse à obtenir le verrou sur les sites S1 et S3 mais que dans le même temps la transaction T2 obtienne le verrou sur les sites S2 et S4. Chaque transaction va attendre d'obtenir le troisième verrou : une situation de « dead lock » est donc présente.

5.4.4.2 Relation d'ordre entre les transactions : schéma d'estampillage

Pour pouvoir assurer une relation d'ordre entre les transactions, le système distribué doit pouvoir posséder un système d'estampille unique.

Il existe principalement deux méthodes pour générer des estampilles uniques dans un système distribué :

- schéma centralisé
- schéma complètement distribué

Une fois que nous pouvons obtenir des estampilles uniques, il est possible de gérer une relation d'ordre entre les transactions. De cette façon, nous pouvons garantir qu'une opération d'écriture ne s'exécute pas avant une opération de lecture émise plus tôt.

5.4.4.2.1 Estampilles uniques

Il existe deux méthodes pour obtenir des estampilles uniques dans un système distribué :

- **Méthode centralisée** : dans ce cas, un site est choisi pour être le serveur d'estampilles. Celui ci peut utiliser un compteur simple ou sa propre horloge locale.

- **Méthode distribuée** : dans ce cas, chaque site génère des estampilles uniques locales. L'estampille peut être un simple compteur ou l'horloge locale. Pour obtenir des estampilles globales uniques, nous concaténons l'estampille unique locale avec le numéro d'identification de la machine (celui-ci doit être unique). L'ordre de la concaténation est important. Le numéro de machine est utilisé comme poids faible de façon à s'assurer que les estampilles générées par une machine ne soient pas toujours supérieures à celles générées par une autre machine.

  Pour résoudre le problème de différence de vitesse de génération d'estampilles entre les machines, il faut un mécanisme de synchronisation des différentes horloges de génération des estampilles locales (algorithme de LAMPORT par exemple).

  Si on utilise l'horloge locale pour la génération des estampilles, nous nous séparons du problème de vitesse d'exécution mais dans ce cas il faut prévoir un mécanisme de synchronisation des horloges physiques des machines du système distribué.

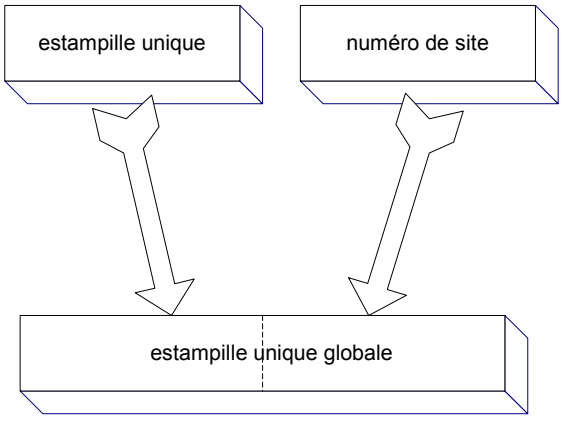


Figure 5.4.4.2.1-1 estampille unique globale

5.4.4.2.2 Schéma d'ordre par estampilles

Pour garantir le respect de l'ordre de génération des transactions, on place les différentes requêtes dans un buffer.

L'analyse du buffer permet de recréer l'ordre de génération des transactions.

Une opération read(x) de Tj doit être différée si il existe une transaction Ti qui doit exécuter une opération write(x) mais ne l'ayant pas encore réalisée. Dans ce cas E(Ti) < E(Tj).

De même, une opération wrtite(x) venant de Tj doit être différée si il existe une opération read(x) venant aussi de Tj.

 Chaque site maintient une liste des opérations différées. Cette technique se nomme **« schéma d'ordre conservateur par estampilles »**.

5.4.5 Détection et gestion des interblocages

5.4.5.1 Définition d'un interblocage

Un système possède un nombre fini de ressources. On entend par ressources, toutes les entités dont les processus ont besoin pour mener à bien leur exécution (RAM, processeur, réseaux, disques durs, unité de sauvegarde, imprimante …).

Un processus doit demander l'autorisation d'accès avant d'utiliser une ressource. La procédure d'utilisation d'une ressource peut se résumer de la façon suivante :

1) Requête : le processus demande au système l'autorisation d'utiliser la ressource. Si la requête ne peut être satisfaite pour l'instant, elle est différée.

2) Utilisation : le processus peut exploiter comme il le veut la ressource.

3) Libération : le processus indique au système qu'il n'a plus besoin de la ressource qu'il vient d'utiliser. Cette libération est aussi appelée « phase de restitution ».

L'étape numéro 2 est appelée « section critique ».

La protection d'accès à une ressource se fait par un mécanisme de verrou. Ce verrou peut se rencontrer sous deux formes :

- Verrou exclusif : dans ce cas, un seul processus, à la fois, peut pénétrer dans sa section critique. On dit qu'il y a « exclusion mutuelle ».

- Verrou partagé : dans ce cas, un nombre fini de processus peuvent pénétrer dans leur section critique en même temps.

Bien entendu, lorsque la ressource ne possède pas, elle même, un verrou (ou en l'absence de moniteurs), on lui associe, par programmation, un verrou dédié (un sémaphore par exemple).


La méthode d'accès décrite plus haut n'est qu'une recommandation à l'attention des programmeurs. Si un programmeur inverse les étapes 1 et 4, le système n'assurera plus une protection d'accès satisfaisante.

Un problème peut survenir avec l'utilisation des verrous d'accès exclusifs :

> Prenez par exemple un système composé de 3 processus gérant 3 unités de sauvegarde. Si chaque processus possède un verrou sur sa propre ressource et qu'il demande à accéder à une des 2 autres ressources, le système sera bloqué. En effet, chaque processus attendra que l'autre libère sa ressource.
>
> Cette situation de blocage du système est appelée **« interblocage de processus »** ou **« dead lock »**.

Tout interblocage est un état indésirable dans un système. Ils sont encore plus durs à prévenir et détecter dans un système distribué que dans un système centralisé.

Par ailleurs, les algorithmes de détection d'interblocage, dans un système distribué, sont assez voisins de ceux que l'on utilise dans un système centralisé.

On peut identifier quatre conditions nécessaires à l'apparition d'un interblocage :

1) **« Exclusion mutuelle »** : une ressource doit se trouver dans un mode non partageable.

2) **« Détention et attente »** : un processus se trouve dans la situation où il détient déjà une ressource et il souhaite accéder à une autre détenue par un processus différent.

3) **« Préemption impossible »** : la ressource possédée par un processus ne peut pas être « préemptée » pour être donnée à un processus plus prioritaire. Seul le processus qui possède la ressource peut décider de la rendre.

4) **« Attente circulaire »** (cycle) : une attente circulaire se définie de la façon suivante : il existe un ensemble de processus $\{P_0, P_1, \ldots, P_n\}$ en attente de ressources tel que P_0 attend une ressource détenue par P_1, P_1 attend une ressource détenue par P_2, …, P_{n-1} attend une ressource détenue par P_n et P_n attend une ressource détenue par P_0.

La détection des interblocages passe par la constitution **« d'un graphe d'allocation de ressources »**. En voici deux exemples :

Le premier graphe montre une situation d'interblocage des processus P1, P2 et P3.

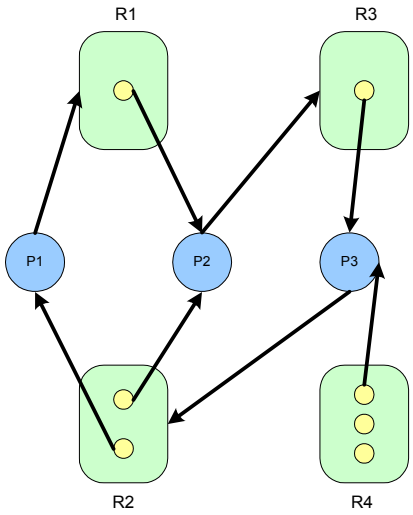


Figure 5.4.5.1-1 graphe d'allocation avec interblocage

Le second graphe montre une situation ou un cycle est détecté mais où la situation d'interblocage n'existe pas si le processus P4 libère la ressource R2.

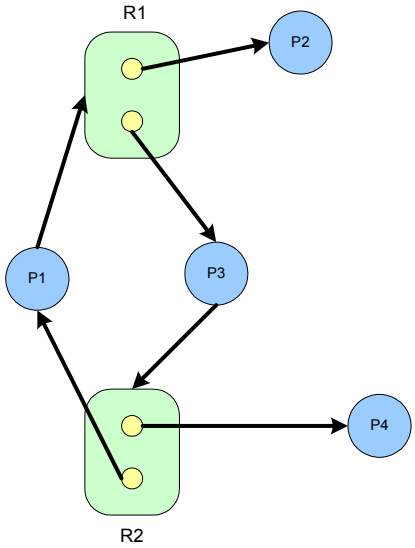


Figure 5.4.5.1-2 graphe avec cycle mais sans interblocage

5.4.5.2 Traitement des interblocages

5.4.5.2.1 Prévention des interblocages

Il existe plusieurs méthodes pour prévenir les interblocages.

La plus simpliste est de déclarer qu'un interblocage ne peut pas se produire ! Cette technique s'appelle « la politique de l'autruche » et elle est à l'origine de bon nombre de problèmes dans des applications informatiques.

5.4.5.2.1.1 Prévention « simple »

Plutôt que de faire des hypothèses sur la probabilité d'apparition d'un interblocage, il est préférable d'employer des algorithmes qui les préviennent.

Parmi ces algorithmes on rencontre « l'algorithme de l'ordre global ». Cette technique consiste à affecter un numéro unique à l'ensemble des ressources du système distribué. Un processus ne peut pas demander une ressource qui possède un numéro supérieur à celle qu'il utilise déjà .

Un autre algorithme consiste à élire un processus comme responsable de l'entretien du graphe d'allocation des ressources du système.

Ces deux schémas sont simples à mettre en œuvre. Le second n'est pas applicable dans un système distribué possédant un nombre important de ressources. En effet, le processus de gestion des graphes d'allocation des ressources devient rapidement un goulot d'étranglement.

Un autre algorithme peut être utilisé pour prévenir les interblocages. Il repose sur la possibilité de préempter les ressources protégées.

5.4.5.2.1.2 Prévention par priorité

Pour contrôler la préemption des ressources, un numéro de priorité unique est affecté à chaque processus du système distribué.

Ces niveaux de priorité sont utilisés pour décider si un processus doit attendre ou non qu'un autre processus libère la ressource attendue.

Prenons deux processus P1 et P2. P1 pourra attendre P2 seulement si sa priorité est supérieure à celle de P2. Dans le cas contraire, P1 sera rétrogradé. Dans cet algorithme, il ne peut pas se produire de cycle. Par contre, il se peut que des processus ne puissent jamais obtenir de ressource. En particulier ceux qui possèdent une priorité faible dans le système.

Pour résoudre ce cas de famine, nous pouvons utiliser des estampilles.

Chaque processus reçoit une estampille unique dans le système (voir le chapitre qui traite de l'attribution des estampilles uniques dans un système distribué).

Deux schémas complémentaires de prévention existent :

1) **Schéma « attendre / mourir »** : lorsqu'un processus Pi demande une ressource détenue par un processus Pj, celui-ci n'est autorisé à attendre que si son estampille est inférieure à celle de Pj. Donc Pi est plus âgé que Pj. Dans le cas contraire, le processus Pi doit être rétrogradé (soit il meurt soit on lui refuse l'accès à la ressource).

2) **Schéma « blessé / attendre »** : cette approche est préemptive car elle autorise un processus à forcer un autre processus à relâcher la ressource qu'il détient. Lorsque le processus P1 souhaite accéder à une ressource détenue par P2, P1 n'est autorisé à attendre que si son estampille est supérieure à celle de P2 (donc P1 est plus jeune que P2). Dans le cas contraire, le processus P2 doit être rétrogradé (on dit que P2 est blessé par P1).

Dans les deux schémas, les cas de famine ne peuvent pas se produire. En effet, puisque les estampilles sont constamment incrémentées, un processus ayant été rétrogradé finira par posséder une estampille lui permettant d'accéder à la ressource.

Par ailleurs, le temps d'attente des processus varie dans un schéma et dans l'autre :

- pour le schéma « attendre / mourir », un processus ancien doit attendre q'un processus plus jeune libère la ressource. Il peut subir plusieurs rétrogradations successives avant d'obtenir la ressource
- dans le schéma « blessé / attendre », un processus ancien peut rétrograder un processus plus jeune. Le processus ainsi rétrogradé sera autorisé à attendre dès qu'il relancera sa demande.

5.4.5.2.2 Détection des interblocages

L'algorithme de prévention des interblocage vu précédemment peut procéder à des préemptions même en l'absence d'interblocage.

Pour éviter un tel cas de fonctionnement, on fait appel à un algorithme de détection des interblocages. Cet algorithme repose sur la construction de graphes d'attente des processus détenant ou souhaitant accéder à des ressources. Chaque site déploie un graphe local.

Le graphe d'attente local contient la liste des processus agissant sur les ressources locales (ceux-ci peuvent être locaux ou distants).

La figure ci-dessous montre deux graphes d'attente situés sur deux sites différents. Les processus P2 et P3 apparaissent à la fois sur le site A et le site B.

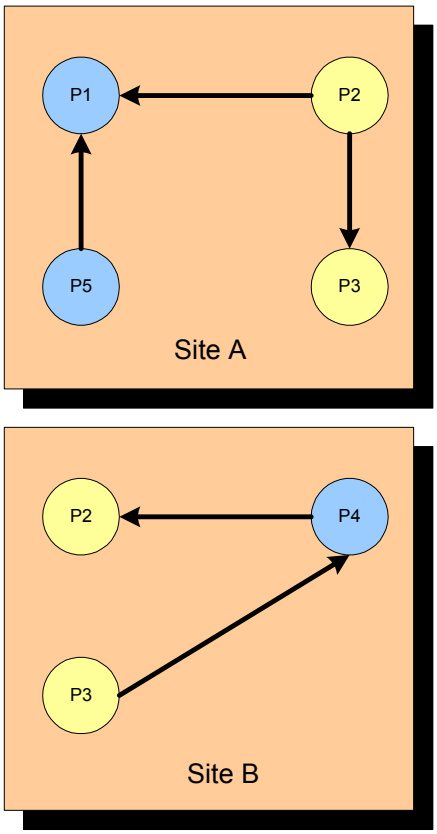


Figure 5.4.5.2.2-1 graphes d'attente locaux

Un cycle dans un graphe local indique la possibilité d'un interblocage.

Par contre, l'absence de cycle dans un graphe local ne peut pas garantir qu'il n'y est pas d'interblocage au niveau de l'ensemble des sites intervenant.

Pour être certain qu'il n'existe aucun cycle dans l'ensemble du système, il faut que la réunion de l'ensemble des graphes d'attente locaux forme un graphe acyclique.

La réunion des deux graphes pris comme exemple fait apparaître un cycle. Cela laisse donc supposer que le système est en état d'interblocage.

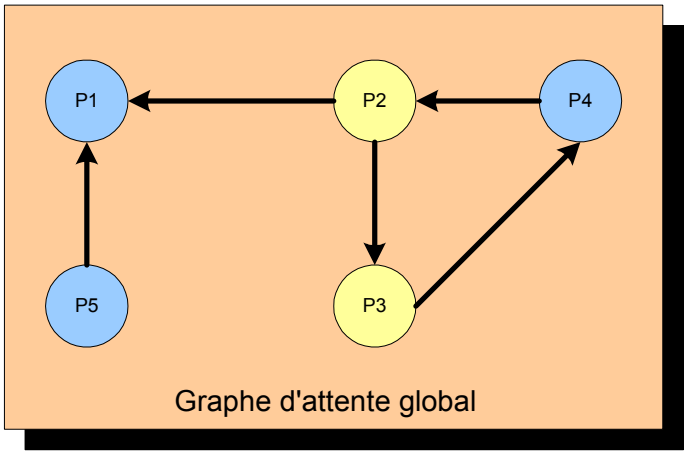


Figure 5.4.5.2.2-2 graphe d'attente global avec un cycle

Cette figure fait donc apparaître la nécessité de gérer un graphe d'attente global dans un système réparti.

5.4.5.2.2.1 Détection des interblocages : méthode centralisée

Dans la méthode centralisée, l'entretien du graphe d'attente global est confié à un processus unique dans le système distribué.

 Ce processus est nommé **« coordinateur de détection des interblocages »**.

Ce graphe global doit signaler tous les interblocages de façon fiable. Pour cela il y a trois méthodes pour mettre à jour la graphe d'attente global :

1) Chaque fois qu'une connexion est ajoutée ou enlevée d'un graphe local, le site concerné envoie un message au coordinateur central pour que celui-ci procède à la mise à jour de son graphe.

2) Périodiquement, après un certains nombre de modifications d'un graphe d'attente local. Cette méthode ajoute un temps de latence dans la détection des cycles.

3) A chaque fois que le coordinateur a besoin d'invoquer l'algorithme de détection de cycle.

Dans tous les cas, il se peut que le coordinateur détecte un cycle et déclenche une phase de récupération (destruction du cycle) alors que le graphe réel ne comporte aucun cycle.

Ce phénomène est lié aux vitesses de transmission des messages dans le réseau et donc à leur ordre d'arrivée.

Pour illustrer ce phénomène, voici les phases de construction du graphe d'attente global pour un système comportant deux sites A et B dont les processus agissent sur des ressources locales et distantes. Le site A a construit un graphe local contenant les processus P1 et P2, le site B a construit un graphe contenant les processus P1 et P3 et le coordinateur central possède un graphe d'attente global contenant les processus P1, P2 et P3. A l'étape 3, le processus P2 a libéré la ressource qu'il détenait dans le site A et demande une ressource détenue par le processus P3 du site B.

Si le message de suppression de la connexion P1→ P2 arrive après celui de connexion P2 → P3, le coordinateur central détecte un cycle alors qu'aucun interblocage ne s'est produit.

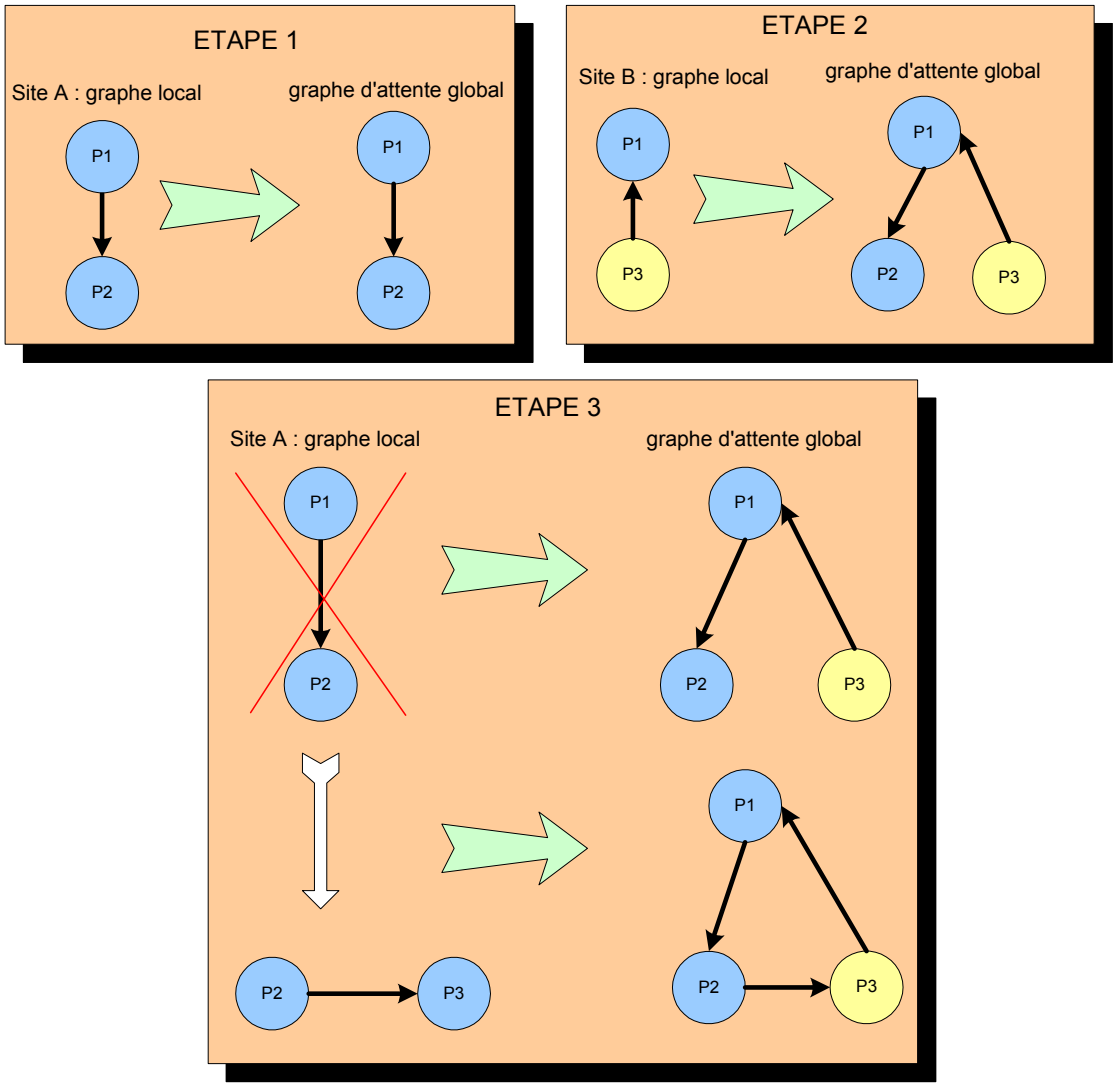


Figure 5.4.5.2.2.1-3 détection d'un faux cycle

Nous avons mis en évidence que l'algorithme utilisé pouvait aboutir à la détection de « faux cycles ».

Pour éviter ce phénomène, les requêtes provenant des différents sites doivent être associées à des estampilles :

> Lorsqu'un processus Pi du site A demande une ressource du site B détenue par P_j, un message associé à l'estampille Es est envoyé a B. Le connexion $P_i \rightarrow P_j$ est insérée dans le graphe du site A. Cette connexion n'est insérée dans le graphe d'attente de B que si le site B ne peut accorder immédiatement la ressource demandée par P_i.
>
> Si une requête $P_i \rightarrow P_j$ est faite en locale (sur un même site) aucune estampille n'est ajoutée.
>
> L'algorithme de détection des interblocages est donc le suivant :
>
> 1) le coordinateur envoie un message demandant au différents sites de fournir leur graphe d'attente locale.
>
> 2) dès réception de la demande du coordinateur, chaque site envoie son graphe d'attente local.
>
> 3) lorsque le contrôleur a reçu tous les graphes d'attente locaux, il construit le graphe d'attente global avec l'algorithme suivant :
>
>   a. le graphe construit contient un sommet pour chaque processus
>
>   b. le graphe possède une connexion $P_i \rightarrow P_j$ si et seulement si il existe une connexion $P_i \rightarrow P_j$ dans l'un des graphes d'attente, ou si une connexion $P_i \rightarrow P_j$ associée à une estampille Es apparaît dans plusieurs graphes d'attente .

5.4.5.2.2.2 Détection des interblocages : méthode répartie

Dans une approche complètement distribuée de la détection des interblocage, tous les sites possède un processus « contrôleur » qui maintient à jour un graphe d'attente local.

Ce graphe local diffère du graphe entretenu dans l'algorithme centralisé. En effet, chaque graphe local peut posséder un nœud supplémentaire lorsqu'un processus fait appel à une ressource externe au site sur lequel il s'exécute. Nous noterons ce nœud « P_{ext} ».

Ce nœud apparaît donc dans deux cas :

- $P_i \rightarrow P_{ext}$: indique que le processus P_i attend une ressource situé sur un autre site
- $P_{ext} \rightarrow P_i$: indique qu'un processus externe au site attend une ressource détenue par le processus P_i

Bien évidemment, la détection des cycles locaux reste valable dans cet algorithme. Si un site détecte un cycle local (ne faisant pas intervenir le processus Pext), le système est déclaré en état d'interblocage et une phase de destruction de cycle doit être entamée. La figure ci-dessous montre le cas d'un cycle local.

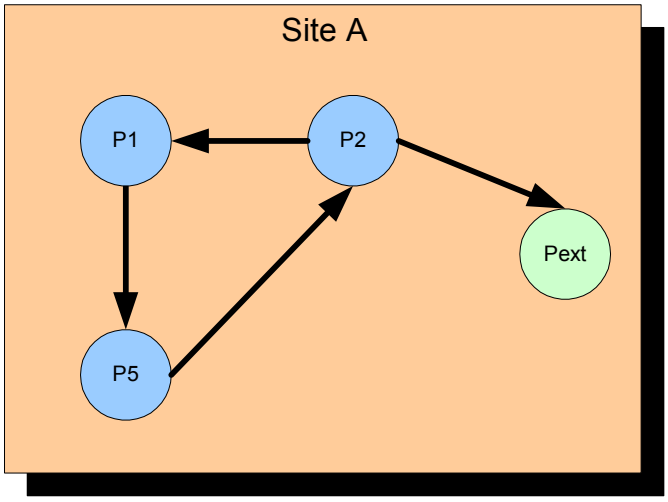


Figure 5.4.5.2.2.2-4 cycle local

Voici un exemple de graphes, sans cycle local, et faisant intervenir P_{ext} .

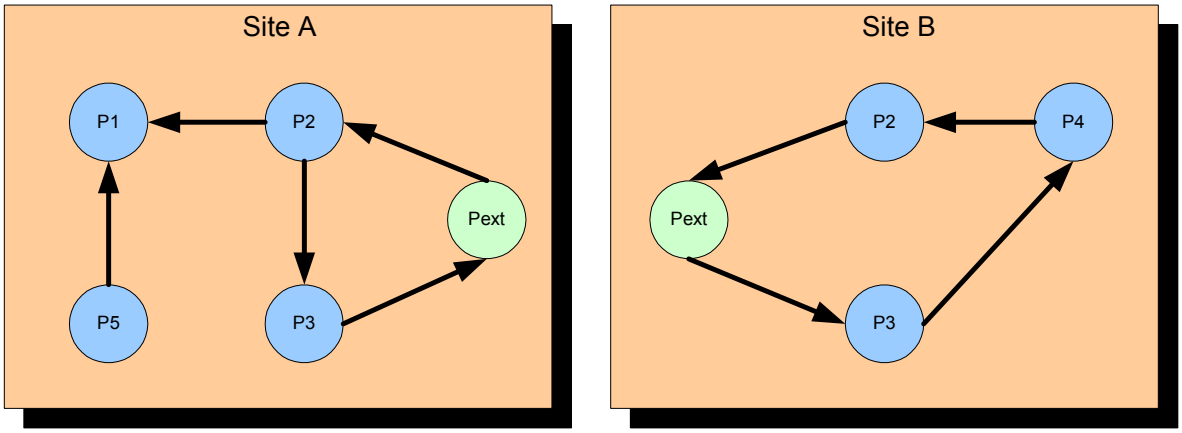


Figure 5.4.5.2.2.2-5 graphes sans cycles locaux

Les graphes locaux présentent des cycles faisant intervenir le processus Pext. Pour autant, il est impossible à l'un des deux sites de savoir si un interblocage est réellement en cours. Chacun n'a qu'une vue partielle du graphe global.

Le site A, détecte cet état et envoie un message au site B lui indiquant le risque d'interblocage. Ce message contient toutes les informations nécessaires à la description du cycle faisant intervenir Pext.

Lorsque le site B reçoit le message, il met à jour son graphe d'attente local avec les informations contenues dans le message de détection d'interblocage.

Ensuite, il lance la procédure de détection des cycles locaux (cycle ne faisant pas intervenir le processus Pext).

Trois cas peuvent se produire :

1) Aucun cycle n'est découvert. La situation d'interblocage n'est pas déclarée.

2) Un cycle n'impliquant pas Pext est découvert. La situation d'interblocage est déclarée et un algorithme de récupération est invoqué.

3) Un cycle impliquant Pext est découvert. Dans ce cas, le site B envoie un message de détection d'interblocage au site concerné par ce cycle. Le site destinataire recommence la même procédure. Après un certains nombre d'itérations, un interblocage existant finira par être découvert.

Pour illustrer cet algorithme, reprenons le système de la figure 5.4.5.2.2.2.

Le site A détecte le cycle Pext → P2 → P3 → Pext. Comme le processus P3 est en attente d'une ressource située sur le site B, le contrôleur du site A envoie un message de détection d'interblocage au contrôleur du site B.

Lorsque le contrôleur du site B reçoit le message de détection d'interblocage, il met à jour son graphe d'attente local.

Le graphe obtenu contient bien un cycle « local ». La situation d'interblocage est déclarée et un algorithme de récupération doit être invoqué.

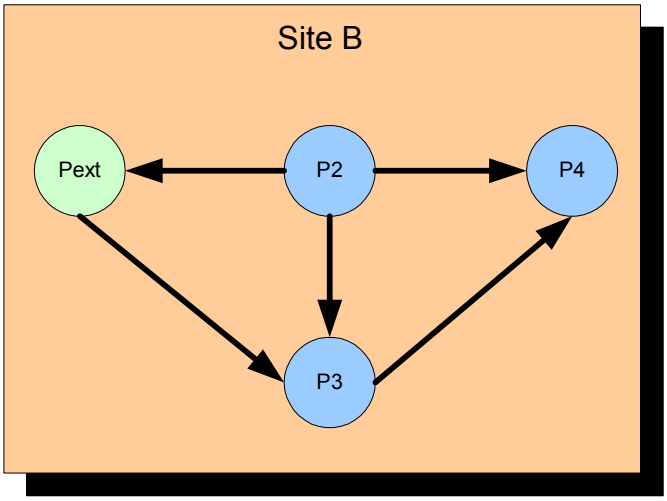


Figure 5.4.5.2.2.2-6 graphe d'attente local étendu

5.4.6 Les algorithmes d'élection

Nous avons dans les chapitres précédents que plusieurs algorithmes utilisent des processus coordinateurs pour résoudre les problèmes liés à la structure distribuée d'un système.

Si le processus coordinateur tombe en panne, le système ne peut continuer à bien fonctionner qu'en procédant à l'élection d'un remplaçant.

Il va de soit que pendant la phase d'élection, le système distribué est dans une situation d'attente de la nomination d'un remplaçant du coordinateur défaillant.

Cette phase d'attente (« latence d'élection ») peut avoir des impacts non négligeables sur le traitement des flux de données répartis. Cette élection va provoquer des « pics de charge » dans tous les sites qui produisent des données (surtout pour des flux isochrones).

Le système doit donc être capable de supporter ce temps de latence. Faute de quoi l'élection d'un remplaçant ne sert plus à rien !

Dans la mesure du possible, il est souhaitable de connaître le temps de latence apporté par une phase d'élection. Cette information nous permet de dimensionner les outils logiciels et matériels d'absorption des pics de charge.

Un des problèmes majeur des algorithmes d'élection est qu'ils ne traitent pas du contexte associé au coordinateur défaillant. C'est à vous de définir une politique de sauvegarde du contexte et de restitution en cas de reprise.

Le contexte peut être dupliqué sur plusieurs sites afin de garantir une bonne tolérance aux pannes (attention au temps de latence supplémentaire au moment de la restitution).

Nous distinguons deux cas de panne différents :

1) **La panne du coordinateur seul** : dans ce cas le site qui l'héberge reste actif. L'élection peut se faire en local : « élection centralisée »

2) **La panne du site qui héberge le coordinateur**. Dans ce cas, l'élection doit déterminer quel site sera utilisé pour lancer une copie du coordinateur : « élection distribuée ».

5.4.6.1 Election centralisée (panne du coordinateur seul)

Si la panne provient du processus lui-même et que le site qui l'héberge est encore actif, l'élection peut se faire en local. Cette élection est assez facile à mettre en œuvre.

Une panne du coordinateur peut être détectée suivant plusieurs politiques :

1) Time-out sur exécution d'une requête. Dans ce cas, un protocole particulier doit être défini pour qu'un processus client puisse interroger le coordinateur sur son état de fonctionnement

2) Une surveillance est confiée à un processus particulier dont le rôle est de s'assurer du bon fonctionnement du coordinateur. Cet agent de surveillance est parfois appelé « FTM » (« Fault Tolerant Manager »).

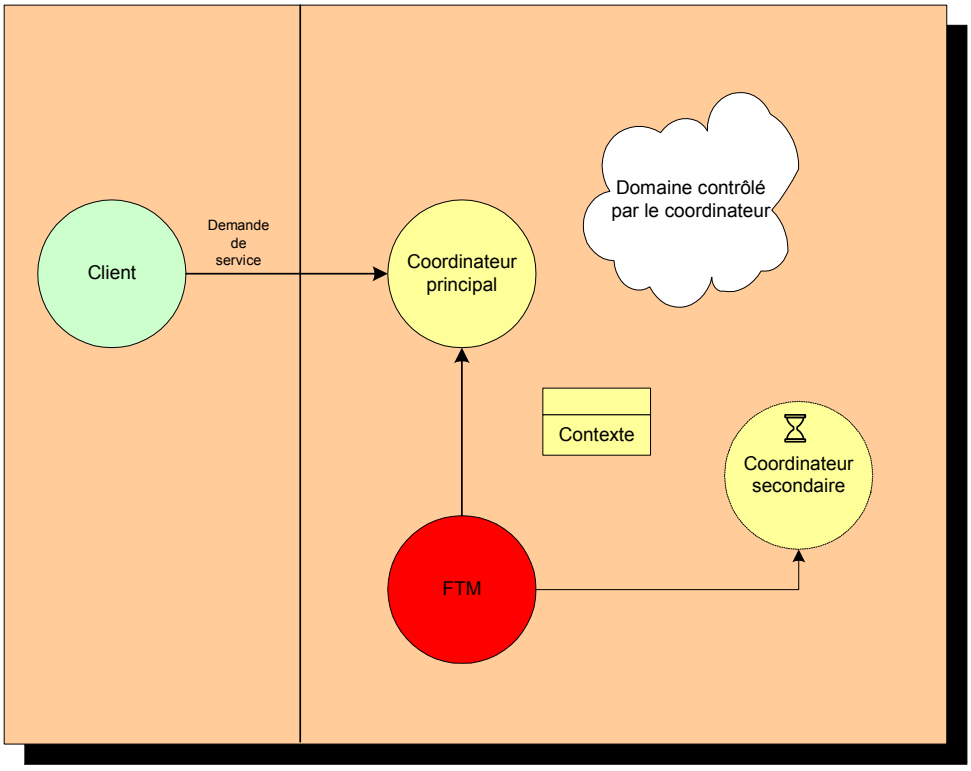


Figure 5.4.6-1 « Fault Tolerant Manager »

En cas de panne du processus coordinateur, une élection doit être entreprise afin de nommer un nouveau coordinateur. C'est le travail du FTM. Celui-ci peut avoir préalablement préparé plusieurs coordinateurs secondaires pour prévoir les cas de défaillance.

Après l'élection, le nouveau coordinateur central doit, bien évidemment, recréer la liste des processus en attente de service.

Dans ce cas, deux algorithmes différents peuvent être utilisés :

1) Interrogation de l'ensemble des processus actifs du système distribué. Cet algorithme sous-entend que le coordinateur a conscience de l'organisation de l'application qu'il sert. On peut aussi utiliser un message en « broadcast » si le système le permet.

2) Récupération de la liste que le précédent coordinateur aura pris soins de sauvegarder à un endroit défini au départ. Cette méthode appelée **« restitution de contexte »** pose un certains nombre de problèmes :

    a. Quel support de stockage pour le contexte ?

    b. Localisation géographique du contexte.

    c. Quelle est la fréquence de sauvegarde du contexte ?

    d. Quelle est la latence de restitution de contexte ?

Dans tous les cas, il se peut que des données soient perdues lors de la défaillance du coordinateur central.

Pour éviter ce phénomène, on utilise la technique de **« duplication de messages »**. Cette technique est aussi appelée **« maître – esclave »**. Un coordinateur est élu « maître » et l'autre « esclave ». Le maître et l'esclave exécute en parallèle les mêmes algorithme mais seul le maître répond réellement à la requête du client.

Un processus client double systématiquement les messages qu'il envoie au coordinateur central. Le premier est envoyé au coordinateur actif (le maître) et le second au coordinateur passif (l'esclave).

En cas de défaillance du maître, l'esclave peut prendre le relais immédiatement et traiter la demande du client. Cette technique masque le temps de latence lié à la phase d'élection.


Cet algorithme demande deux fois plus de message que nécessaire mais assure qu'aucune perte de donnée ne peut intervenir.

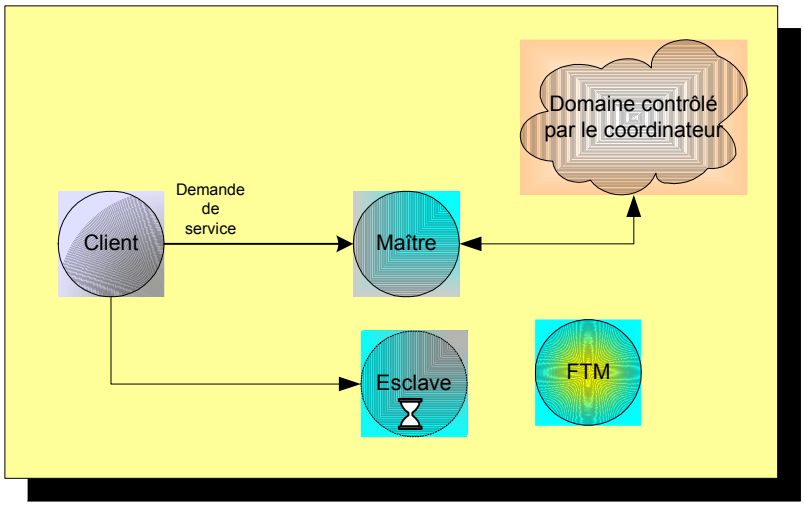


Figure 5.4.6-2 structure maître - esclave

5.4.6.2 Election distribuée : l'algorithme par priorité

Cet algorithme fonctionne dans un système où les processus distribués reçoivent tous un numéro de priorité unique.

Supposons qu'un processus Pi émette une requête à destination du coordinateur central. Si il ne reçoit aucune réponse dans un délais (défini au préalable), Pi considère que le coordinateur est en panne et va donc provoquer une nouvelle élection.

Pi informe (il envoie un message) tous les processus de priorité supérieure à la sienne qu'il se présente comme le successeur du coordinateur défaillant.

Il attend, pendant un temps « T », une réponse de l'un des processus informés de l'élection en cours.

Deux cas peuvent se produire :

1) Pi ne reçoit aucune réponse des processus qu'il a informé de sa candidature. Dans cette situation, il lance une copie du coordinateur central et informe l'ensemble des processus, de priorité inférieure à la sienne, du résultat de cette élection. Il, n'est pas utile d'informer les processus de priorité supérieure car ceux ci sont considérés en panne puisqu'ils n'ont pas répondu.

2) Pi reçoit une réponse d'un des processus de priorité supérieure. Dans ce cas, il attend de nouveau pendant un délai « T' » un message l'informant de la réussite de l'élection. Si aucun message ne parvient pendant le délai « T' », Pi considère que le processus qui avait répondu est en panne. Il recommence alors une nouvelle élection.

Les processus du système peuvent donc recevoir, à tout moment, l'un des deux messages suivants :

- Le processus Pi est le nouveau coordinateur.

- Le processus Pi a commencé une phase d'élection. Dans ce cas, la priorité du processus qui reçoit ce message est forcément supérieure à celle de Pi. Il répond donc à Pi et commence une phase d'élection.

Le processus qui achève son algorithme est autorisé à lancer une copie du coordinateur. Il informe l'ensemble des processus de priorité inférieure du numéro du nouveau coordinateur.

Dans le cas ou un coordinateur de priorité maximum dans le système se récupère, il informe tous les processus qu'il devient le nouveau coordinateur. C'est pour cette raison que cet algorithme est aussi appelé **« algorithme brutal »**.

5.4.6.3 Election distribuée : l'algorithme de l'anneau

L'algorithme de l'anneau suppose deux conditions :

1) Tous les processus sont connectés ensemble par anneau logique unidirectionnel. Ils envoient un message à leur seul voisin de droite et reçoivent un message de leur seul voisin de gauche.

2) Les processus possèdent tous un numéro de priorité unique dans le système.

Le but recherché est que chaque processus construise en local une liste des différents processus existants dans le système. Le classement se fera par ordre de priorité.

Supposons que le processus Pi détecte une défaillance du coordinateur central. L'algorithme de l'anneau est alors le suivant :

1) Pi crée une liste vide des processus présents dans le système. Il y ajoute sa priorité « i » et envoie le message « élire (i) » à son voisin de droite.

2) Lorsque Pi reçoit un message « élire (…) » de son voisin de gauche, il doit réagir de trois façons différentes :

    a. S'il s'agit de son premier message « élire(j) » vu, Pi crée une nouvelle liste locale et y insère les priorités reçues ainsi que la sienne (la priorité de son voisin de gauche). Il envoie ensuite le message « élire (i, j) » à son voisin de droite.

    b. Si le message reçu ne contient pas la priorité de Pi, Pi ajoute les numéros à sa liste et fait suivre le message à son voisin de droite après avoir ajouté son propre numéro.

    c. Si le message reçu contient la priorité de Pi, Pi est certain de connaître (au travers de sa liste locale) l'ensemble des processus présents dans le système et donc le processus possédant la plus forte priorité. Il connaît donc le processus qui doit procéder à une relance d'un coordinateur central.

        i. Pour avertir les autres processus de l'anneau que l'élection a abouti, Pi génère le message « R(k, …., s) ». Chaque processus qui reçoit ce message, enlève son numéro de la liste reçue et fait suivre le reste de la liste à son voisin de droite.

Cet algorithme permet, en deux tours de l'anneau, de procéder à une élection et d'informer l'ensemble des processus de la constitution de l'anneau.

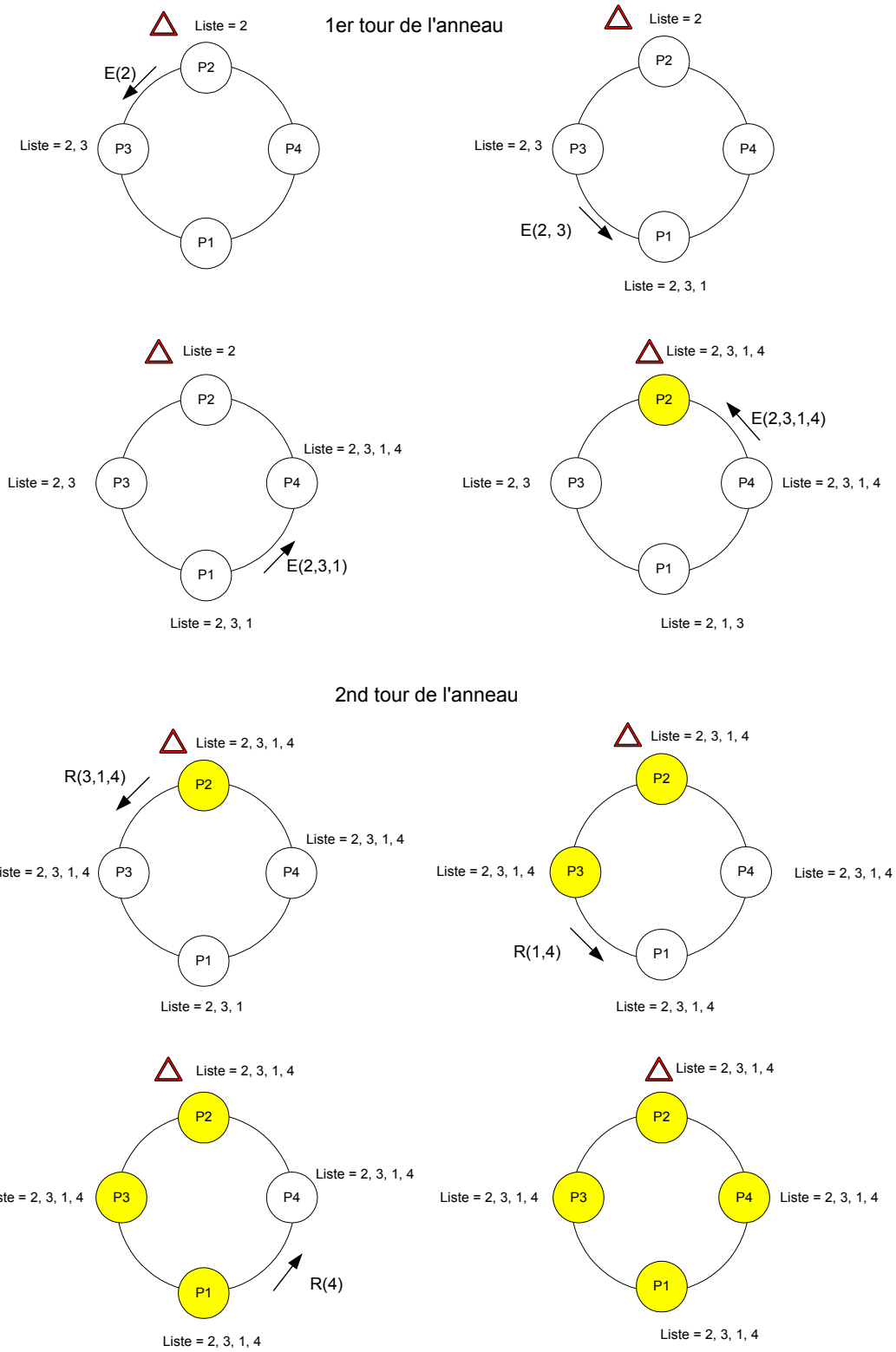


Figure 5.4.6.3-1 algorithme de l'anneau

5.5 Systèmes de fichiers distribué

Les programmes informatiques manipulent des fichiers destinés à rendre persistantes les données traitées. Ces données peuvent être des informations de paramétrage, des résultats de calculs, des traces …

Dans un système distribué, il est intéressant de posséder un système de fichiers distribué sur le réseau de façon à répartir l'information sur différents sites.

Les différents systèmes de fichiers distribués se regroupent sous le terme **DFS** (« Distributed File System »).

Dans l'idéal, un système de fichiers distribué doit apparaître aux clients comme un système de fichiers centralisé.

Le DFS doit savoir localiser les fichiers d'un client et rendre transparent le transport des données lié aux différentes manipulations effectuées sur ces fichiers (lecture, écriture, copie …).

Sur un système de fichiers centralisé, le temps d'accès à un fichier est essentiellement lié au support physique de stockage (le temps de traitement par le processeur local est négligeable face au temps d'accès des disques durs actuels).

La performance d'un DFS, en terme de temps de réponse, est liée au réseau d'interconnexion des différents sites d'un système distribué. Pour chaque requête de lecture ou d'écriture, le système local subit le temps de traitement des différentes couches du ou des protocoles utilisés pour distribuer l'information sur le réseau.

5.5.1 Nommage dans un système de fichiers distribué

Comme dans un système centralisé, l'utilisateur d'un système distribué manipule des fichiers en utilisant un nom. Le DFS doit donc masquer à l'utilisateur, la localisation géographique des fichiers manipulés.

On distingue deux notions importantes dans la transparence d'accès aux fichiers :

- transparence de l'emplacement : le nom du fichier ne comporte aucune information sur la localisation physique du fichier manipulé
- indépendance de l'emplacement : si le fichier change de localisation géographique, le fichier continu d'être accessible à l'utilisateur de façon transparente.

La plupart des DFS supportent la transparence de l'emplacement mais pas l'indépendance de l'emplacement d'un fichier.

5.5.1.1 Politique de nommage

Il existe deux approchent principales de politique de nommage dans un DFS.

La première approche consiste à associer l'identifiant de la machine d'hébergement au nom du fichier. Par exemple, le système QNX permet de manipuler des fichiers distribués sur le réseau en précisant le numéro de nœud ou se trouvent les fichiers. Cette politique ne permet pas de transparence de l'emplacement des fichiers néanmoins les entrées/sorties se font avec les mêmes fonctions sur un fichier local ou distant.

La seconde approche consiste a permettre d'attacher des répertoires distants à une arborescence locale. De cette façon, l'accès aux fichiers se fait comme ci ceux-ci étaient présent sur la machine locale. Cette approche a été implémentée dans le protocole NFS (« Network File System ») de la société SUN. Des techniques « d'automontage » permettent d'améliorer l'indépendance de l'emplacement des fichiers. Les répertoires sont montés à la demande de façon automatique et d'après des tables de montages.

5.5.2 Accès à des fichiers distants

Lorsqu'un utilisateur procède à des accès sur un fichier distant, le DFS peut utiliser deux méthodes pour répondre.

5.5.2.1 Accès directes par RPC

La première méthode consiste à produire des appels de procédures distantes pour chaque requêtes d'accès. Par exemple, un client produit un appel à la fonction « read() » : la machine d'exécution transforme cet appel en une trame RPC à destination du serveur. Celui effectue la lecture des secteurs physiques concernés et retourne les données lues au client.

Cette technique implique des appels RPS pour chaque requêtes de lecture et d'écriture distantes. Le réseau subira donc une surcharge dans le cas d'accès répétés aux fichiers distants.

5.5.2.2 Accès par l'intermédiaire de mémoires cache

La seconde méthode d'accès à des fichiers distants repose sur l'utilisation de zones de mémoire cache.

Le système de mémoire cache disque d'un système centralisé n'a pour seul but que d'augmenter la vitesse d'accès aux fichiers situés sur un disque dur. Le système de mémoire cache disque d'un système distribué a un autre but : celui de minimiser le trafic réseau.

Le principe du cache est de maintenir en mémoire locale une copie partielle du fichier distant. Un accès sur une zone de fichier déjà présente dans le cache se fera directement en mémoire vive (sans surcharge liée au réseau). Si il n'existe pas de copie locale de la zone manipulée, le client demandera au serveur de lui fournir une les données correspondantes et les placera dans sa mémoire cache.

Afin de limiter la taille du cache, des algorithmes ressemblant à ceux de la gestion de mémoire virtuelle sont mis en place (LRU par exemple).

Lorsqu'un cache est modifié, les modifications doivent être reportées sur le serveur et sur l'ensemble des machines qui manipulent la zone modifiée. La gestion de la cohérence des caches est délicate et doit être fiable pour éviter les pertes de données.

La mise à jour des caches peut être faite soit à l'initiative du serveur soit à l'initiative des clients.

5.5.2.2.1 Politiques de mise à jour des caches

Il existe plusieurs politiques de mise à jour du cache d'un système de fichiers distribué.

La méthode la plus simple se nomme « l'écriture double ». Lorsqu'un client procède à une écriture dans la mémoire cache, le système envoie immédiatement les données à écrire au serveur distant. Cette méthode améliore la fiabilité du système de fichier distribué puisqu'en cas de panne, seul le cycle d'écriture en cours est perdu.

Par ailleurs, l'utilisation de cette méthode revient à n'utiliser la zone de cache que lors des accès en lecture.

Une seconde méthode consiste a différer les écritures des fichiers modifiés. Cette technique permet d'éviter les échanges de données répétés entre le client et le serveur dans le cas ou le fichier subit plusieurs accès en écriture dans un intervalle de temps court. Cette politique d'écriture différée impose des pertes de données importantes en cas de panne d'une machine.

Le transfert des données à écrire sur le serveur peut être fait suivant différentes méthodes. Par exemple :

- Lorsqu'un bloc est sur le point d'être supprimé du cache local. Cette méthode limite considérablement le trafic réseau. Par contre, certains blocs de fichiers peuvent rester longtemps en mémoire cache avant d'être mis à jour sur le serveur.

- A une fréquence donnée : le processus chargé de la gestion du cache local vient vérifier quels sont les blocs ayant subit une modification à intervalle régulier. Les blocs modifiés sont transférés vers le serveur pour la mise à jour. NFS utilise cette méthode.

Une autre politique consiste à n'écrire les zones de cache que lors de la fermeture du fichier. Cette méthode donne de bons résultats sur des fichiers ouverts pour de longues périodes et modifiés fréquemment. En cas de défaillance, la perte de donnée peut être conséquente. De plus, la fermeture d'un fichier devient une opération longue puisqu'elle doit attendre que l'ensemble des blocs situés en mémoire cache soient transférés sur le serveur distant.

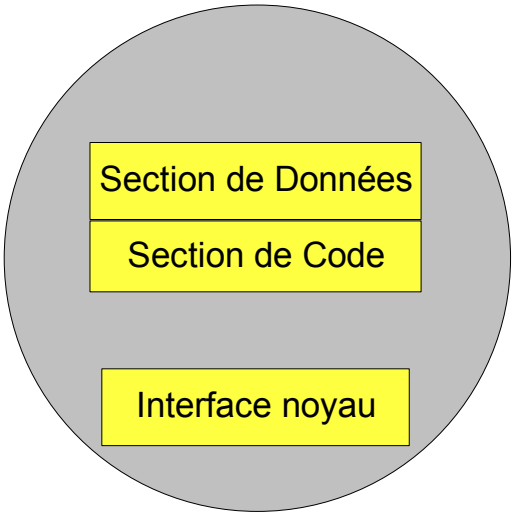
6. ANNEXES

6.1 Rappels sur le multitâche centralisé

6.1.1 Les différences entre processus et threads

6.1.1.1 Le processus

- Un processus est aussi appelé HWP (Heavy Weight Process) ou processus de “ poids lourd ”. Sa structure est la suivante :



- Un processus possède une section de code et une section de données (et une section de pile qui peut se confondre avec la section de données)

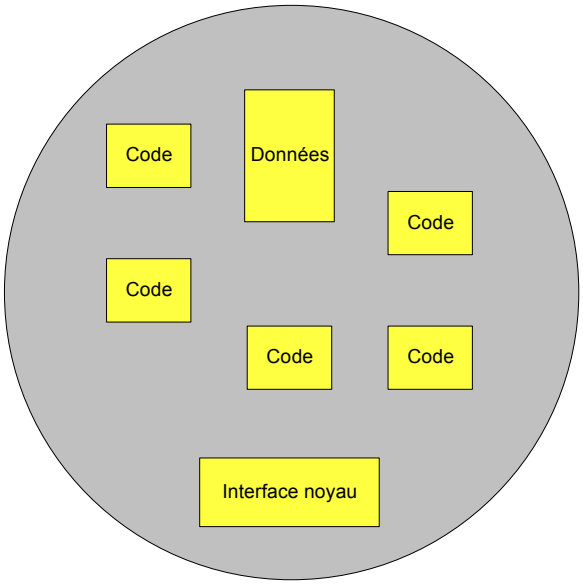
6.1.1.2 Le thread

- Un thread est aussi appelé LWP (Light Weight Process) ou processus de “ poids léger ”

- Il est constitué d’un compteur d’instructions, d’un ensemble de registres et d’un espace de pile

- Un thread partage avec des threads ressemblants sa section de code et de données ainsi que les ressources du système d’exploitation comme les fichiers ouverts et les signaux

- Un ensemble de thread est connu du système d’exploitation comme un processus

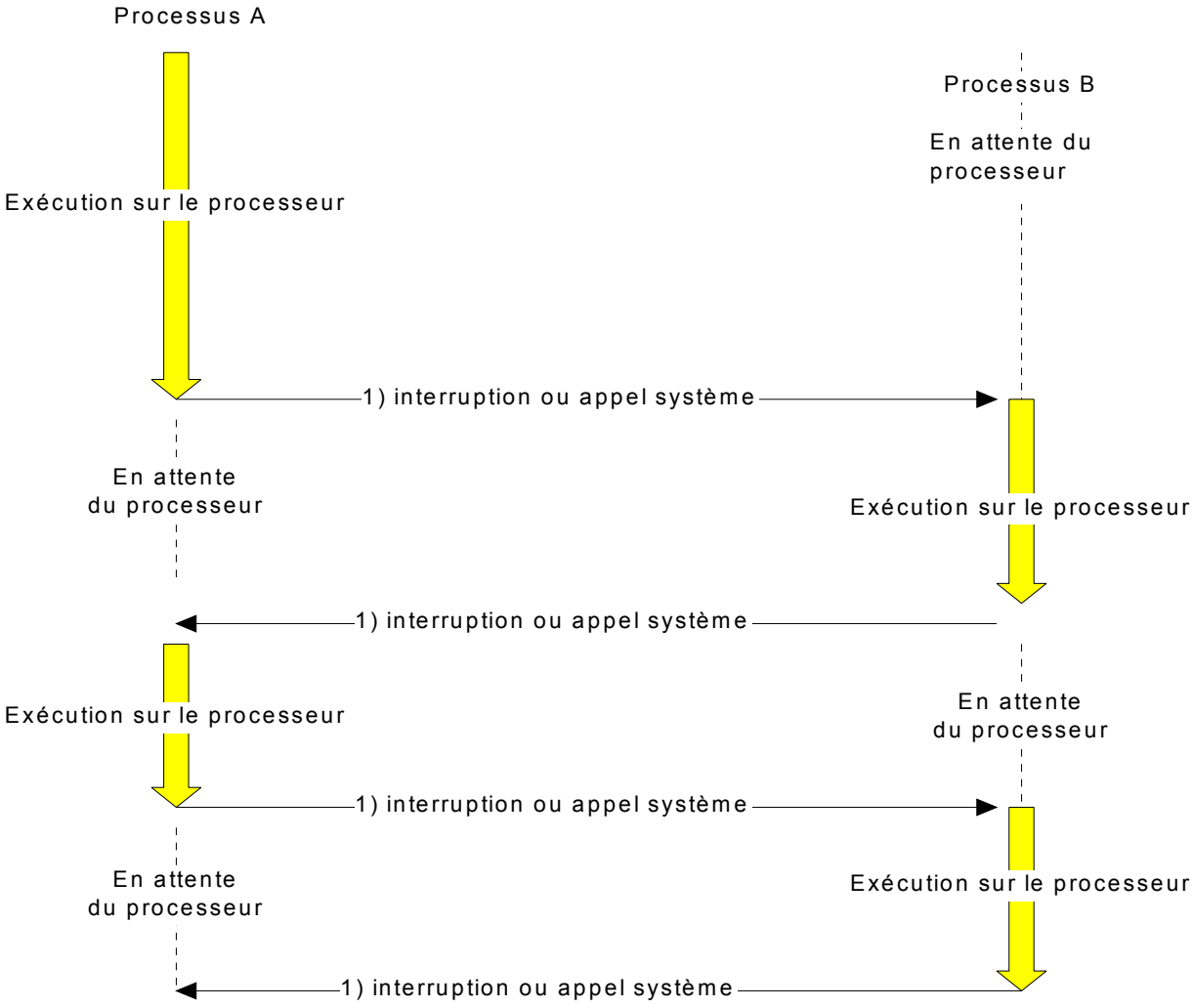


- Le fait que les threads partagent leur section de code et de données ainsi que les ressources systèmes rend la commutation de contexte entre les threads ainsi que la création de thread peu coûteuses en cycle machine

6.1.2 Commutation de processus dans un système multitâche centralisé

- L'objectif d'un système multitâche est de commuter le processeur entre les processus suivant l'arrivée d'événements, la priorité des tâches, le démarrage ou la terminaison de processus

- Dans un système monoprocesseur, il n'y aura jamais plus d'un processus en exécution. S'il existe plus d'un processus, ceux-ci devront attendre jusqu'à ce que la CPU soit libre et puisse être « schédulée » de nouveau



6.1.3 La préemption

- On parle de préemption lorsqu'un processus en cours d'exécution sur la CPU doit immédiatement être remplacé par un autre traitement plus prioritaire (autre processus, interruption …)

- La préemption d'un processus par un autre processus suit les règles décrites dans le chapitre traitant des algorithmes de scheduling

- La préemption du système d'exploitation est la capacité du noyau à interrompre un traitement en cours lorsqu'il est lui-même en exécution sur la CPU (exemple : interruption d'un appel système)

- Pour assurer la « préempitivité » d'un OS, le code du noyau doit être « réentrant »

6.1.3.1 Algorithme de scheduling du processeur

- Quand les processus rentrent dans le système, ils sont placés dans une file d'attente de travaux. Cette file d'attente contient tous les processus du système

- Les processus résidants en mémoire principale qui sont prêts et attendent de s'exécuter sont maintenus dans une liste appelée « file d'attente des processus prêts » (ou ready queue). Cette liste d'attente est généralement implémentée sous la forme d'une liste chaînée

- La ready queue contient des pointeurs vers les TCB des différents processus

- Le scheduling est régi par différents types d'algorithmes qui peuvent se rencontrer en même temps sur un même système d'exploitation

6.1.4 Communication et synchronisation entre les processus

6.1.4.1 Sémaphore

- Un sémaphore est un objet logiciel utilisé pour :
  - assurer l'exclusion mutuelle (partage de ressources)
  - assurer la synchronisation de plusieurs processus
- Les sémaphores sont gérés entièrement par le système d'exploitation centralisé.
- **Section critique** : ce terme désigne la section de code s'exécutant entre la décrémentation et le l'incrémentation d'un sémaphore.

- Le partage des ressources se fait avec un sémaphore appelé **« sémaphore de comptage »**. Ce sémaphore peut prendre des valeurs de 0 à « n ». « n » indique le nombre d'accès simultanés que peut supporter la ressource. Tant que « n » est supérieur à 0, la fonction « Prendre() » peut s'exécuter. Le système d'exploitation décrémente « n » et le processus peut continuer à s'exécuter sur le processeur. Lorsque le sémaphore vaut zéro, l'appel à la fonction « P() » à pour effet de mettre le processus en attente jusqu'à ce qu'un autre processus incrémente le sémaphore (appel à la fonction « V() »)

- La synchronisation de processus ou la protection d'accès à une donnée partagée par au moins deux processus se fait avec un sémaphore appelé **« sémaphore binaire »**. Ce type de sémaphore ne peut prendre que les valeurs 0 ou 1. Il faut faire attention à la valeur d'initialisation de ces sémaphores.

- A un instant donné, plusieurs processus peuvent être en attente sur un sémaphore (sémaphore = 0). C'est le système d'exploitation qui décide quel est le processus qui doit être « réveillé » lorsque le sémaphore prend une valeur différente de zéro. Ce choix est fait en fonction de l'algorithme de « scheduling » ainsi que de la priorité des processus

- Prenons le cas où un ou plusieurs processus sont en attente sur un sémaphore valant zéro. Si ce sémaphore n'est jamais incrémenté, on dit alors que l'on se trouve dans un cas de « famine ». Dans ce cas, l'application cesse de fonctionner correctement

❑ Prenons le cas où deux processus utilisent deux sémaphores que l'on appellera « sem1 » et « sem2 ». Les deux processus exécutent les opérations suivantes :

Processus n° 1

Décrémenter(Sem1) ← début de la première section critique de P1

Attendre (1 seconde) ← mise en sommeil du processus P1

Décrémenter (Sem2) ← début de la seconde section critique de P1

Traitements divers … SECTION CRITIQUE …

Incrémenter(Sem2) ← fin de la seconde section critique de P1

Incrémenter (Sem1) ← fin de la première section critique de P1

Processus n° 2

Décrémenter (Sem2) ← début de la première section critique de P2

Attendre (1 seconde) ← mise en sommeil du processus P2

Décrémenter (Sem1) ← début de la seconde section critique de P2

Traitements divers … SECTION CRITIQUE …

Incrémenter (Sem1) ← fin de la première section critique de P2

Incrémenter (Sem2) ← fin de la seconde section critique de P2

Lorsque le processus P1 est mis en sommeil pendant 1 seconde, le processus P2 peut prendre le sémaphore n° 2 puis se mettre à son tour en sommeil pendant 1 seconde. Le processus P1 se réveille avant le processus P2 et tente alors de prendre le sémaphore n° 2. Celui-ci étant déjà pris par le processus P2, P1 est alors mis en attente de l'incrémentation (« Incrémenter (sem2) ») du sémaphore n° 2 par le processus P2. Le processus P2 se réveille à son tour et tente de prendre le sémaphore n° 1 qui est déjà pris par le processus P1. A cet instant, le système se trouve dans un état bloqué. Cet situation d'inter blocage de processus est appelée **« dead lock »** (ou étreinte mortelle).

6.1.4.2 File interprocessus

- Une file est un composant logiciel fonctionnant en FIFO (First In First Out)
- Un « pipe » est un exemple de file
- Si une file à plusieurs producteurs ou consommateurs, il faut synchroniser les accès entre producteurs et consommateurs

6.1.4.3 Pile interprocessus

- Une pile est un composant logiciel fonctionnant en LIFO (Last In First Out)
- Tous les processus possèdent une pile destinée à gérer les appels de fonctions, les interruptions, les variables manipulées, …

6.1.4.4 Queue de messages : communication interprocessus « asynchrone »

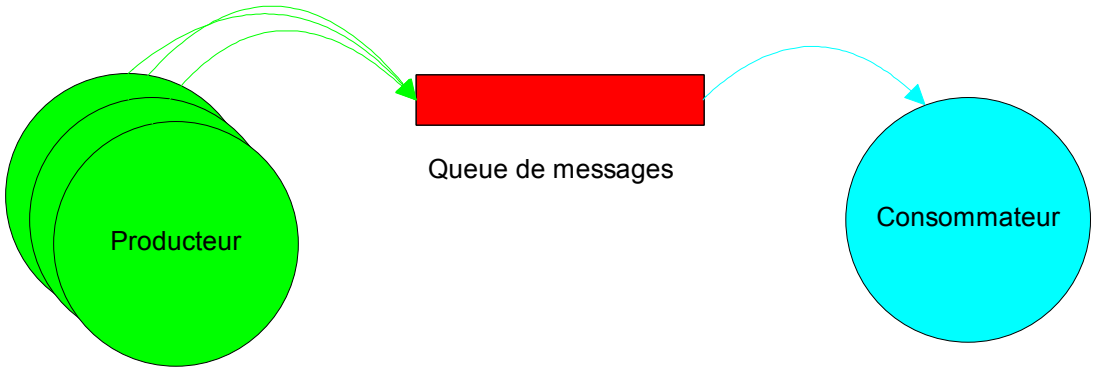
- Une queue de messages est un mode de communication inter tâche dit « asynchrone »

- Les queues de messages sont des boîtes aux lettres pouvant recevoir de 1 à « n » messages

- Les queues de messages sont utilisées dans le cas ou le consommateur ne s'exécute pas à la même fréquence que le producteur.

- Le dimensionnement des queues de messages est très important pour la bonne exécution des applications multitâche

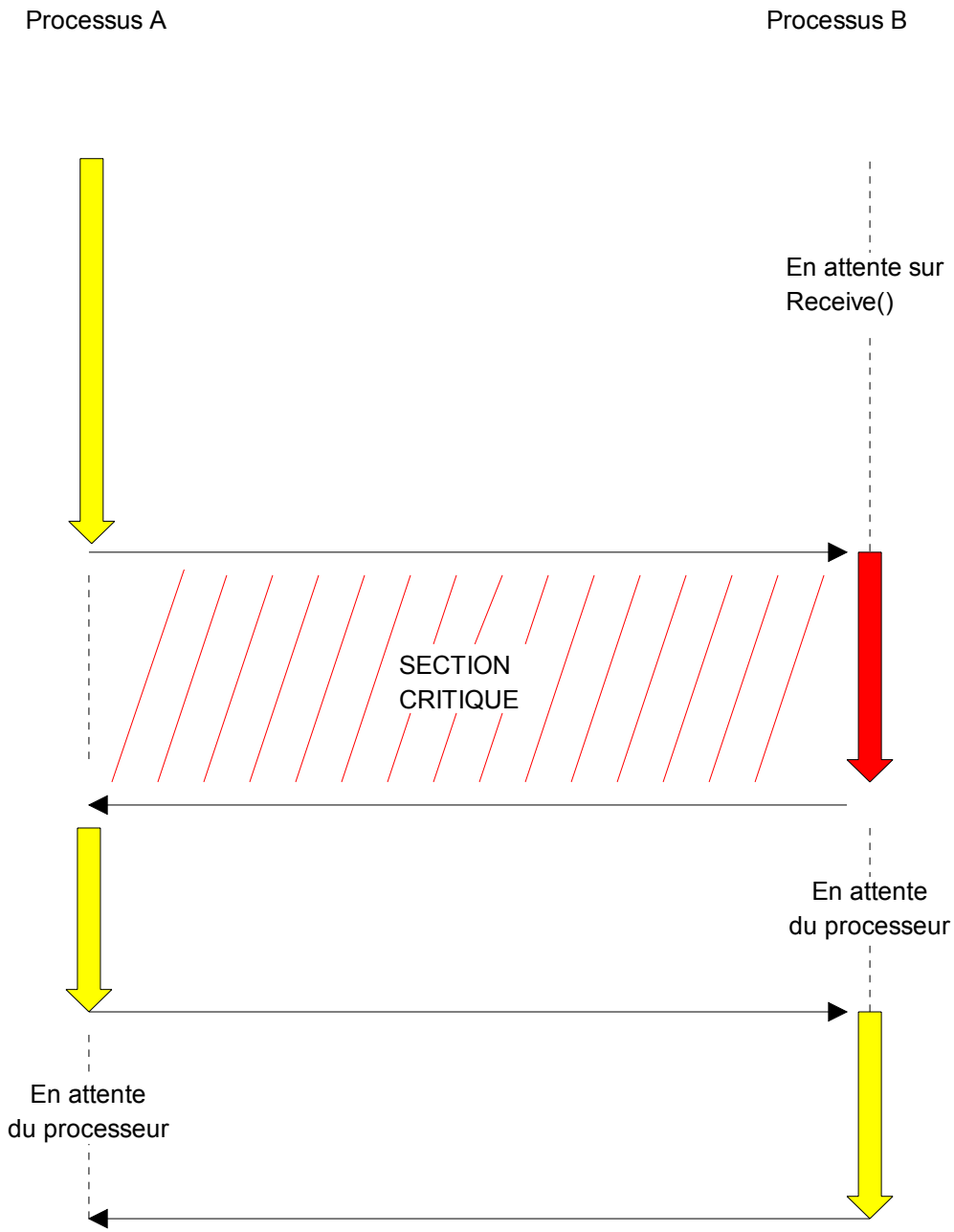
- On peut avoir plusieurs producteurs et plusieurs consommateurs sur une même queue de message. Dans la pratique, il y a « n » producteurs et un seul consommateur. Ex :



6.1.4.5 Communication interprocessus synchrone (« message passing »)

- La communication, interprocessus, par messages est un mode dit « synchrone »
- Les micros noyaux utilisent ce mode de communication par défaut pour dialoguer avec leurs différents managers
- La communication par message est aussi appelée « message passing » dans la littérature anglo-saxone
- La communication par message impose des sections critiques
- Plusieurs processus peuvent envoyer chacun un message vers un autre processus « en même temps »
- Le « message passing » est le seul mode de communication qui traverse le réseau dans les systèmes d'exploitation distribués
- Le « message passing » est très rapide car le système d'exploitation n'utilise pas de buffers intermédiaires pour transférer un message d'un processus vers un autre

- schéma d'une communication par message :



Ce schéma varie suivant la priorité des processus A et B

6.1.4.6 Signaux

- Les signaux normalisés par POSIX :
  - Les signaux sont utilisés pour la synchronisation de processus
  - Ils représentent des événements atomiques ne portant aucune donnée
  - Seulement deux signaux, SIGUSR1 et SIGUSR2 sont disponibles pour le développement spécifique
  - Les signaux POSIX ne fonctionnent pas dans une architecture distribuée

6.1.4.7 Communication par la mémoire

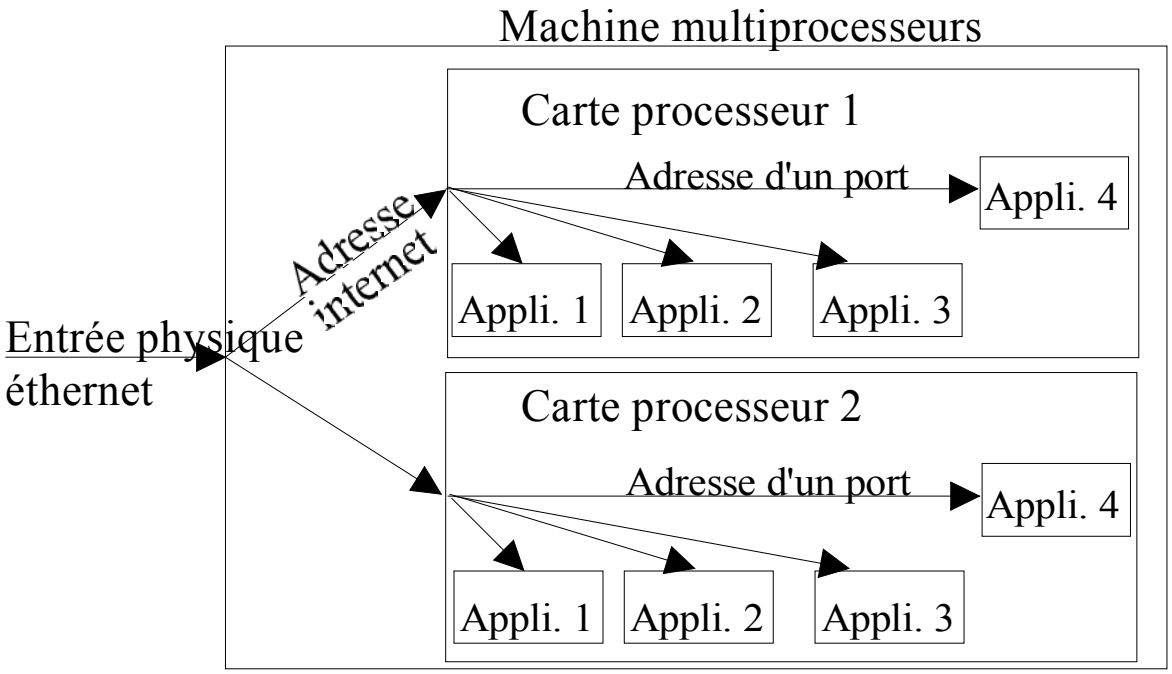
- Dans les systèmes ne fonctionnant pas en mode « protégé », la mémoire principale de la machine peut être directement utilisée pour communiquer entre les processus

- Dans les systèmes fonctionnant en mode protégé, seuls les segments de mémoire partagée doivent être utilisés pour échanger des données entre différents processus *(SIGSEGV…)*

- La communication à travers la mémoire auxiliaire est possible mais est extrêmement lente. Elle est utilisée pour des transmissions de gros volume de données

6.2 La gestion des sockets (TCP/IP)

- Les sockets sont des points d'entrée logiciel sur un réseau.
- Une tâche d'une machine ouvre une socket pour communiquer avec une autre tâche sur une autre machine.
- Ils forment des connexions point à point.
- On peut ouvrir plusieurs sockets entre deux machines.
- Les sockets utilisent un niveau d'adressage supplémentaire. En effet, Internet permet d'adresser une machine, au mieux une carte dans une machine. Pas une tâche ou un programme tournant dans une machine.
- Ce niveau supplémentaire est appelé un port.



6.2.1 Principe

- Les sockets sont liées à la notion client/serveur. Un serveur est en attente d'une connexion. Le client demande une connexion.

- Un serveur utilise deux sockets. L'une pour recevoir la demande de connexion. Une autre, qu'il crée, pour la réaliser. Ce mécanisme lui permet de recevoir plusieurs connexions.

- Le fonctionnement peut être comparé au téléphone.

Serveur	Téléphone	Client
Crée une socket	ouvre une ligne	
Se met à l'écoute	attend	
	numérote	Appelle (adresse internet)
	demande un poste	N° du port
Accepte l'appel	décroche	
Lit/écrit	communication	lit/écrit

- A partir de ce moment là on est en communication normale. Le premier qui raccroche rompt la communication.

- Une fois la liaison établie on peut utiliser les commandes read/write de base du C.

- Les sockets peuvent fonctionner en deux modes:
  - STREAM : Garantit l'arrivée correcte des packets.
  - DATAGRA : Transmet les packets sans les gérer.

Important: Une socket transmet des octets. Elle ne tient aucun compte, ni aucune trace des limites d'enregistrements;

6.2.2 Initialisation d'un serveur

Voici le code source en langage « C » d'un serveur de connexion par socket.

```
/* ouverture d'une socket */

Target.sin_family = AF_INET;
Target.sin_port = PORT_NUMBER;
Target.sin_addr.s_addr = INADDR_ANY ;

SocketId = socket(AF_INET, SOCK_STREAM, 0);
if (SocketId < 0)
        Error("\nserver2: Erreur sur appel de socket ") ;

/* obtention de l'adresse internet */

if (bind(SocketId, (struct sockaddr *)&Target, sizeof(Target))
        Error("\nserver2: Erreur au bind ");

/* ecoute */
if (listen(SocketId, 1))
        Error("\nserver2: Erreur sur listen");

/* Boucle de traitement */

while(1)
{
        /** attente d'une connexion **/
        if ((NewSocket = accept(SocketId, (struct sockaddr*)0,0)) == -1)
        Error("\nserver2: Erreur sur accept");

                /**** Autres traitements ****/
}
```

6.2.3 Initialisation d'un client

Voici le code source en langage « C » d'un client de connexion par socket.

```
/* ouverture d'une socket */

Target.sin_family = AF_INET;
Target.sin_port = PORT_NUMBER;
Target.sin_addr.s_addr = 0;

SocketId = socket(AF_INET, SOCK_STREAM, 0);
if (SocketId < 0)
Error("client: Erreur sur appel de socket ") ;

/* obtention de l'adresse internet */

if ( (TargetDesc = gethostbyname(HOST_SERVER)) == NULL )
Error("client: Erreur sur appel de gethostbyname ");

bcopy ((char *)TargetDesc->h_addr, (char *)&Target.sin_addr,
                        TargetDesc->h_length);

/** essai de connexion **/

if (connect(SocketId, (struct sockaddr *)&Target, sizeof(Target)) = -1)
        Error("\nclient: Erreur de connexion ");

/** Suite du traitement **/
```

A partir de ce moment là le comportement du serveur et du client sont identiques.

6.2.4 Ecriture / lecture

Elles peuvent se faire avec les commandes read / write classiques du C:

```
if (write(SocketId, Buffer, TailleBuffer) == -1)
    {
        Error("client: erreur d'emission \n");
    }

if (read(NewSocket, Buffer, TailleBuffer) == -1)
    {
        Error("server: Erreur en reception\n") ;
    }
```

Ou avec des commandes spécifiques:

```
if (send(SocketId, (char*)&TailleBuffer, sizeof(TailleBuffer), 0) == -1)
    {
        Error("client: erreur d'emission \n");
    }

if ((NbRecu = recv(socket, pt, Manque, 0)) == -1)
    {
        return (NbRecu) ;
    }
```

6.2.5 Précaution particulière

Ce n'est pas clair dans les documentations et jamais dans les exemples:

Les fonctions read et recv ne garantissent pas le nombre d'octets reçus

Elles rendent la main dès qu'elles ont reçu quelque chose, ce qui pose des problèmes si les messages envoyés sont supérieurs à une trame (1500 octets).

Il FAUT vérifier le nombre d'octets reçus

```
int SII_receive(int socket, unsigned char *pt, int size)
{
  int NbRecu, TotalRecu = 0, Manque = size ;

  while (Manque > 0)
  {
     if ((NbRecu = recv(socket, pt, Manque, 0)) == -1)
     {
        return (NbRecu) ;
     }
     pt += NbRecu ;
     Manque -= NbRecu ;
     TotalRecu += NbRecu ;
  }
  return (TotalRecu) ;
}
```

7. Bibliographie

- Principes des systèmes d'exploitation : Abraham Silberschatz et Peter B. Galvin aux éditions Addison-Wesley
- L'informatique répartie : Michel Gabassi, Bertrand Dupouy aux éditions Eyrolles
- La programmation sous UNIX : Jean-Marie Rifflet aux éditions Ediscience
- La communication sous UNIX, applications réparties : Jean-Marie Rifflet aux éditions Ediscience
- Les systèmes d'exploitation, structure et concepts fondamentaux aux éditions Masson