UNIVERSITÉ DE
SHERBROOKE

Département d'informatique
Faculté des sciences

IFT 630 - Processus concurrents et parallélisme

Chapitre 7

Systèmes dustribués

Gabriel Girard[1]

Sherbrooke

14 mars 2023

[1] Gabriel.Girard@usherooke.ca

Table des matières

Chapitre 7

Systèmes multiprocesseurs et répartis

Les systèmes mono-processeur sont généralement très limités en puissance de calcul. Quand un système contient plus d'un processeur, la puissance de calcul est décuplée permettant ainsi de traiter des problèmes complexes, auparavant impossibles à résoudre sur un mono-processeur.

7.1 Classification des architectures matérielles

Commençons par classifier les systèmes possédant plusieurs processeurs :

- Systèmes fortement couplés (mémoire commune)

  Cette architecture est composée de plusieurs processeurs (ou cœurs) qui partagent la même mémoire centrale et les mêmes unités d'entrées/sorties. Ces processeurs sont sous le contrôle intégré d'un seul système d'exploitation.

- Systèmes faiblement couplés (réseau)

  Cette architecture est composée de systèmes relativement autonomes, chacun possédant son propre système d'exploitation, sa propre mémoire centrale et ses propres unités d'entrées/sorties. Les différents systèmes sont reliés par un réseau et sont parfois appelés à se coordonner pour l'exécution de certaines tâches.

- Systèmes avec des processeurs fonctionnellement spécialisés

  Avec cette architecture, on retrouve communément un ou plusieurs processeurs dits «généraux» combinés avec des processeurs spécialisés. Ces derniers sont contrôlés par les processeurs généraux et leur fournissent des services en exécutant certaines tâches.

  Par exemple, on retrouve fréquemment des ordinateurs comprenant des cartes graphiques spécialisées (Nvidia ou autres). De même, il arrive qu'un système possède d'autres types de cartes spécialisées, comme certaines pour les entrées/sorties ou toutes autres fonctionnalités.

7.1.1 Les systèmes fortement couplés (mémoire commune)

Les systèmes fortement couplés sont extrêmement populaires. Nous les retrouvons autant dans les ordinateurs que dans les cellulaires/tablettes.

Cette architecture est très efficace et offre une très bonne augmentation des performances. Toutefois, son principal inconvénient est les accès à la mémoire centrale par les différents processeurs qui provoquent rapidement un goulot d'étranglement.

En effet, rappelons que tous les processeurs accèdent fréquemment à la mémoire centrale pour y récupérer les instructions et les données. Les opérations impliquant la mémoire centrale sont tellement fréquentes qu'elles deviennent problématiques lorsqu'il y a plus d'une dizaine de processeurs. Trois architectures mémoires ont été proposées dans l'industrie du multiprocesseurs pour palier à ce problème : UMA (Uniform Memory Access), NUMA (Non Uniform Memory Access) et COMA (Cache Only Access Memory). La répartition des ressources matérielles varie d'un modèle à l'autre ce qui affecte les performances lors de l'accès. De plus, selon [108, 47], la bande passante disponible dans l'architecture UMA est limitée car elle emploie un seul contrôleur de mémoire. Le principal motif de l'avènement des machines basées sur l'architecture NUMA est d'améliorer la bande passante disponible pour la mémoire en exploitant plusieurs contrôleurs de mémoire. La figure 7.1 illustre la différence entre les architectures UMA et NUMA.

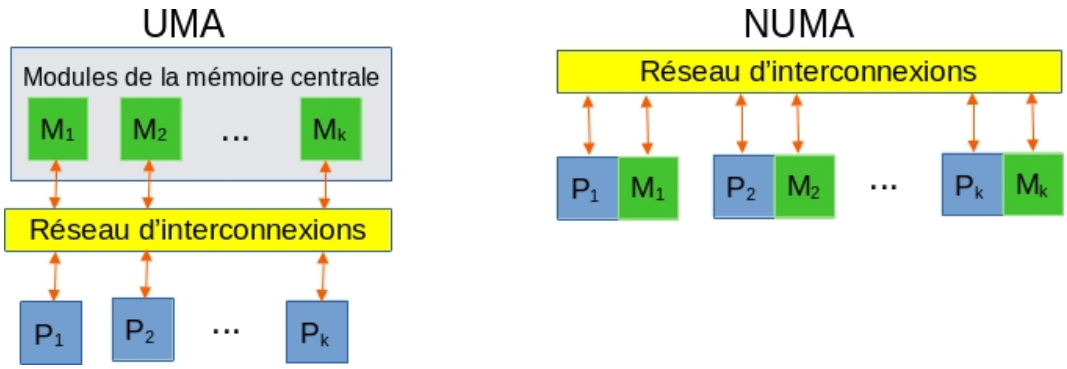

Figure 7.1 – Les architectures UMA et NUMA

Architectures UMA

L'architecture UMA [10, 116, 108, 47] est une organisation fournissant à tous les processeurs un accès «uniforme» à la mémoire centrale . Cela signifie que le temps ainsi que la vitesse des accès à toutes les cellules de la mémoire sont les mêmes pour tous. Comme elle fournit un accès équilibré à la mémoire, cette architecture est celle généralement choisie dans les architectures symétriques, parfois appelée (faussement ?) SMP (multiprocesseur symétrique). Elle pose toutefois des problèmes de mise à l'échelle. En effet, lorsque le nombre de processeurs augmente, la communication entre la mémoire et les processeurs devient un goulot d'étranglement et les temps d'accès en souffrent. Pour permettre une meilleure mise à l'échelle, on associe souvent à chaque processeur de la mémoire cache.

Un réseau d'interconnexions assure l'accès à la mémoire. Plusieurs technologies ont été proposées dans l'espoir de remédier à ce problème de goulot d'étranglement. Les principales technologies pour

implanter ce réseau sont :

- Bus unique

  Cette technologie, simple et très populaire, illustrée à la figure 7.2, fournit un chemin commun que tous doivent emprunter en même temps pour accéder à la mémoire centrale. Étant donné que le bus est une ressource partagée, cette technologie possède une capacité d'expansion plutôt limitée (le bus devient rapidement un goulot d'étranglement lorsque le nombre de processeurs augmente). Cette technologie est reconnue pour supporter aisément de 2 à 4 processeurs mais difficilement de 32 à 64 processeurs.

  La solution couramment employée pour augmenter le nombre de processeurs sur un bus est de réduire les accès en fournissant de la mémoire cache. Celle-ci est associée à chaque processeur (L1 - dans le CPU), à chaque sous-ensemble de processeurs (L2 - près des CPU) et parfois à tous les processeurs à la fois (L3 - sur la carte). La figure 7.2 présente un exemple d'architecture basée sur un bus combiné avec de la mémoire cache associée à chaque processeur.

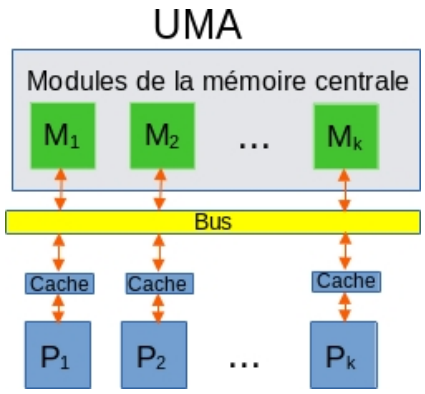

Figure 7.2 – Un modèle UMA utilisant un Bus et une cache

  Même l'ajout de mémoire cache ne résout pas entièrement le problème car son efficacité est conditionnelle au fait que les accès à la mémoire soient très localisés. De plus, on doit gérer des problèmes de cohérence. Ainsi, le bus risque tout de même de redevenir rapidement un goulot d'étranglement. Selon Fienup [34, 49], ce type d'architecture ne supporte que de 2 à 36 processeurs.

- un commutateur à barres croisées («crossbar switch») [49]

  Pour augmenter la bande passante et espérer supporter plus de processeurs, une nouvelle technologie d'interconnexion, appelée commutateur à barres croisées ou «crossbar switch», a été proposée. Cette dernière connecte indépendamment n processeurs à k modules de mémoire. Selon cette approche, on sépare la mémoire en «N» modules, où «N» correspond au nombre de processeurs. Ainsi, pour 32 processeurs, la mémoire est séparée en 32 modules distincts. Avec cette organisation, tous les processeurs ont la capacité d'accéder simultanément à la mémoire en autant que les modules de mémoire accédés soient distincts. La figure 7.3 présente un exemple de commutateur à barres croisées connectant huit processeurs à huit modules de mémoire.

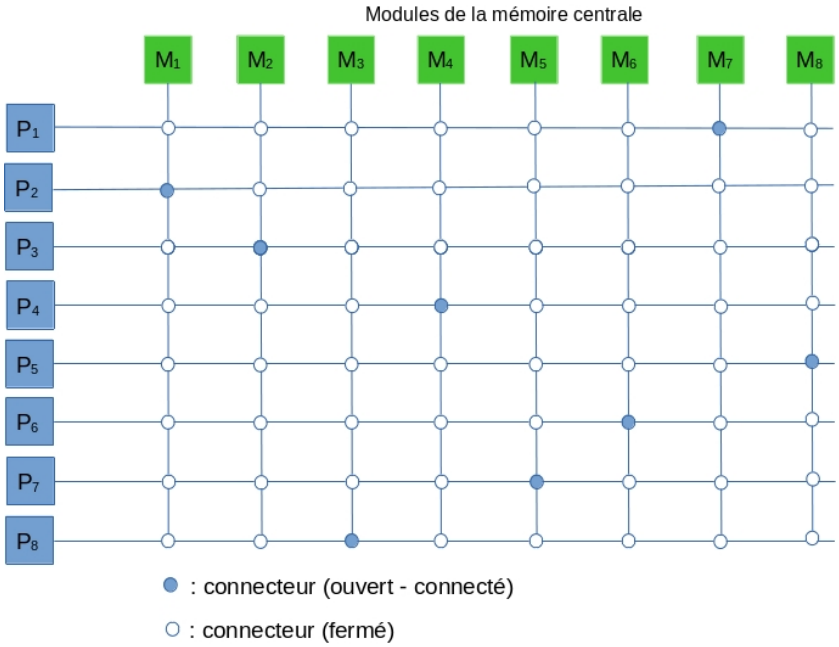

Figure 7.3 – Une commutateur reliant 8 processeurs et 8 modules de mémoire

Le problème avec cette technologie est qu'elle requiert une très grande quantité de connexions, soit N^2 connexions en présence de N processeurs. Pour huit processeurs et autant de modules de mémoire, on a besoin de 64 connexions. Pour 1000 processeurs, cela nécessitera 1000×1000 connexions.

En conclusion, cette technologie s'avère coûteuse et ne permet pas vraiment de supporter beaucoup plus de processeurs. Selon Fienup [34], ce type d'architecture supporte de 2 à 256 processeurs.

- un réseau de connecteurs multi-étapes («multistages switching networks»).

  Ce type de réseau utilise plusieurs niveaux de connecteurs pour diminuer le nombre de connexions possibles. La figure 7.4 présente ce type de technologie. Dans cet exemple, 12 connecteurs relient huit processeurs à huit modules de mémoire (versus 64 pour le «crossbar»). Ce type de réseau supporte toutefois moins d'accès concurrents à la mémoire que le commutateur barres croisées car les connecteurs sont partagés et ne peuvent servir deux processeurs en même temps. Il nécessite toutefois beaucoup moins de connexions. Au final, cette technologie ne supporte pas beaucoup plus de processeurs que les deux précédentes techniques.

Architecture NUMA

Cette architecture [10, 116] a pour objectif de résoudre les problèmes de capacité d'expansion du modèle UMA. Selon cette architecture, la mémoire est divisée en autant de modules qu'il y a de processeurs et chacun des modules est associé à un processeur. Chaque processeur possède donc

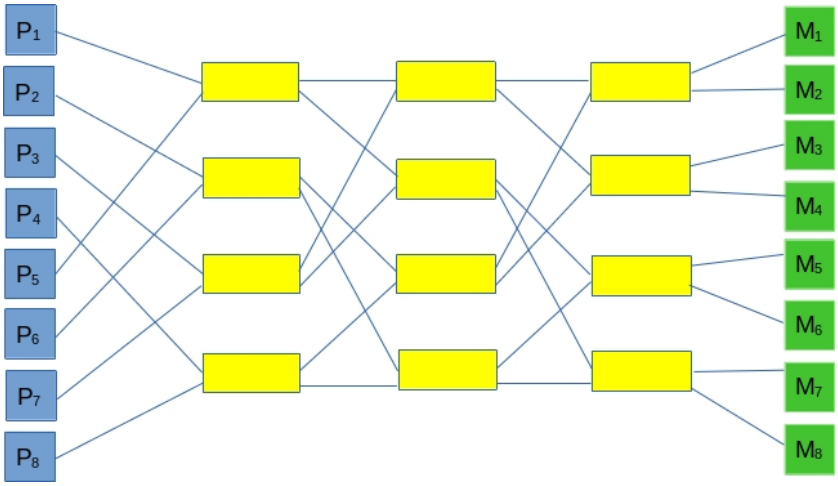

Figure 7.4 – Une «multi-stage crossbar switch» reliant 8 processeurs et 8 modules de mémoire

de la mémoire locale et de la mémoire éloignée tel qu'illustré aux figures 7.5 et 7.6. La mémoire est tout de même organisée comme un seul espace d'adresses mais le temps et la vitesse d'accès varient selon les adresses visées. L'accès à une adresse mémoire locale est très rapide, tandis que l'accès à une adresse mémoire «éloignée» est plus lente. De cette non-uniformité vient le nom de NUMA.

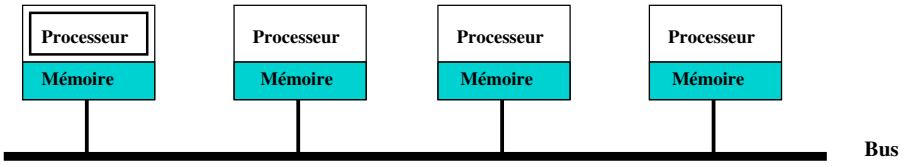

Figure 7.5 – Structure de la mémoire NUMA

Cette architecture est transparente pour les usagers. Qu'il soit local ou éloigné, l'accès à une adresse mémoire se fait par l'intermédiaire des instructions «`load`» et «`store`». Toutefois un «`load`» ou un «`store`» sur de la mémoire éloignée est beaucoup plus lent. Pour compenser ce fait concernant les accès éloignés, on choisit parfois une architecture dérivée, appelée ccNUMA, où le «cc» représente le recours à une mémoire cache.

Selon Fienup [34], ce type d'architecture est en mesure de supporter de 2 à 256 processeurs.

COMA (Cache-only Memory Access)

Il y a peu de chose à dire sur l'architecture COMA [114], si ce n'est que c'est une forme de NUMA dans laquelle la mémoire locale est continuellement employée comme une mémoire cache.

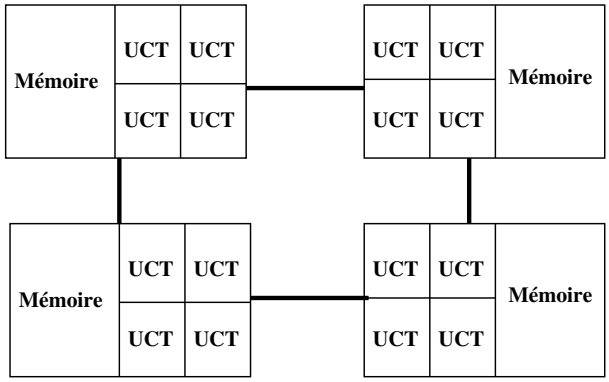

Figure 7.6 – Structure de la mémoire NUMA

7.2 Classifications des systèmes d'exploitation

Les systèmes d'exploitation ont dû être adaptés afin de gérer les systèmes fortement couplés (multi-processeurs) ou les systèmes faiblement couplés (multi-ordinateurs). Il existe donc différents «types» de systèmes d'exploitation selon leur architecture sous-jacente.

7.2.1 Systèmes d'exploitation pour les multiprocesseurs

Les systèmes dits multi-processeurs sont conçus pour gérer de multiples cœurs sur une même puce ou de multiples processeurs dans le même ordinateur. L'élément commun important est qu'ils doivent tous partager la même mémoire centrale.

La conception de systèmes d'exploitation aptes à gérer de multiples processeurs présente de nombreux défis. C'est pourquoi plusieurs sous-types de systèmes ont été proposés au fil des années.

Chaque processeur (UCT) possède son propre système d'exploitation

Selon cette organisation, chaque UCT possède sa propre copie privée du système. La mémoire de l'ordinateur est divisée en autant de partitions qu'il y a de copies. Il n'y a donc aucun problème de synchronisation car chacun possède ses propres copies privées de chacune des structures de données du système (liste des processus prêts, ...).

Il n'y a donc aucun partage de processus et peu ou pas de partage de mémoire. Cela implique que si un processeur devient surchargé, il est impossible de transférer des tâches vers d'autres processeurs.

Cette architecture ne permet pas de tirer profit au maximum de la présence de plusieurs processeurs. De ce fait et de plusieurs autres problèmes, cette organisation est rarement privilégiée.

Une organisation de type «Administrateur/Travailleurs»

Selon l'organisation «Administrateur/Travailleurs», un seul processeur, l'administrateur, exécute le système d'exploitation. Les autres processeurs (travailleurs) ne maintiennent que l'environnement nécessaire à l'exécution des tâches qui leur sont confiées par l'administrateur. Ils ne s'occupent donc que de l'exécution des programmes.

Selon cette architecture, il y a partage de processus et de mémoire. Il y a peu de problèmes de synchronisation car l'administrateur gère toutes les structures du système.

Le hic avec cette organisation est que l'administrateur risque rapidement de devenir un goulot d'étranglement. En effet, il doit gérer tous les appels systèmes et toutes les entrées/sorties. 20 processeurs suffisent généralement à le surcharger. De plus, l'administrateur est un unique point de panne.

Cette architecture fut populaire au début des environnements multi-processeurs mais elle est abandonnée aujourd'hui.

Multiprocesseurs symétriques

L'organisation couramment utilisée sur tous les multiprocesseurs modernes consiste en un système multiprocesseurs symétriques.

Selon cette approche, un seul système d'exploitation est partagé par tous les processeurs. Toutes les structures du système ainsi que la mémoire sont partagées. Cela rend possible un balancement de charge dynamique (processus peuvent changer de processeur). Cependant la synchronisation est nécessaire pour contrôler l'accès aux différentes structures (mutex, verrous, ...). La granularité de la synchronisation est importante afin d'obtenir une performance optimale.

Par exemple, les noyaux de Linux (avant 2.6) définissaient le noyau au complet comme une section critique. Ainsi chaque appel au système s'exécutait en exclusion mutuelle. Cette granularité très grossière était très coûteuse. En effet une vingtaine d'UCTs suffisaient à provoquer une surcharge dans le système.

Il est donc nécessaire de diviser le noyau du système d'exploitation en plusieurs sections critiques indépendantes. Cette division s'est avérée l'une des tâches les plus complexes dans la conception de systèmes multi-processeurs. En effet, un conception inadéquate est susceptible de provoquer un interblocage.

Synchronisation de systèmes multiprocesseurs

Comme déjà mentionné, il s'avère impossible de synchroniser des tâches s'exécutant sur différents processeurs en bloquant les interruptions. La seule technique pour synchroniser de multiples processeurs est celle de l'attente active basée sur des instructions spéciales exécutant de façon atomique deux accès à la mémoire centrale (lecture/écriture ou écriture/écriture). Pour assurer l'atomicité des deux opérations, le processeur verrouille le bus d'accès à la mémoire. Aujourd'hui, tous les bus (et les protocoles qu'ils fournissent) autorisent le verrouillage des accès [1].

Comme on a recourt de l'attente active, ces instructions provoquent une lourde charge de travail sur le processeur et sur le bus (tentative continuelle en boucles pour lire ou écrire). Pour éviter cette surcharge, plusieurs pistes de solutions ont été proposées [104] :

- Utilisation d'une mémoire cache

  Cette solution ne fonctionne pas car une instruction de type «`testAndSet`» effectue une écriture qui invalide la cache ;
- Lecture suivie d'un test.

1. Si jamais un bus ne permettait pas de verrouiller les accès, il faudrait avoir recourt à un algorithme d'attente active comme celui de Peterson.

Il est possible de faire une lecture préalable, puis d'exécuter l'instruction «`tstAndSet`» seulement si le verrou est disponible. Dans ce cas précis, la présence d'une mémoire cache est profitable.

- Interrogation avec délai

  Plutôt que de boucler sans interruption en interrogeant constamment la mémoire, il est possible d'imposer un délai entre chaque interrogation. Ce délai s'allonge généralement entre chaque essai infructueux.

- Utilisation d'une file d'attente.

- Changement de contexte

  Dans certaines situations, il s'avère préférable de faire un changement de contexte plutôt que d'attendre activement. Cela n'est pas toujours souhaitable cependant. **Comment décider s'il est mieux d'attendre activement ou de suspendre l'exécution ? Quelles sont les conséquences de chacune ?** Concernant l'attente active, elle provoque une consommation continue et inutile de temps UCT. Le changement de contexte, lui, entraîne aussi une consommation de temps UCT mais pour effectuer le changement de contexte lui-même. Ainsi, il y a perte de temps pour le système à accomplir les tâches suivantes :

  – sauvegarder l'environnement du processus (registres, ...) ;
  – modifier la cache (pour charger les instructions et les données du nouveau processus) ;
  – modifier la TLB (fautes de pages) pour charger le nouveau processus ;
  – modifier la cache (pour recharger les instructions et les données de l'ancien processus) ;
  – modifier la TLB (fautes) pour recharger l'ancien processus ;
  – récupérer l'environnement (registres, ...)

  Supposons que le temps «perdu» pour un changement de contexte est d'environ 2 ms (1 ms pour changer vers le nouveau processus et 1 ms pour retourner à l'ancien). Ainsi, si l'attente dure environ 50 µs, il est alors plus efficace de faire de l'attente active. Si au contraire, l'attente dure en moyenne plus de 2 ms, il est alors plus avantageux de procéder à un changement de contexte.

  Des études semblent indiquer que la meilleure approche consiste à d'abord vérifier/mesurer les temps d'attente, pour ensuite établir/configurer la politique.

7.2.2 Systèmes d'exploitation pour les multi-ordinateurs

Pour gérer un réseau d'ordinateurs, il existe deux grandes familles de systèmes d'exploitation :

- systèmes d'exploitation réseaux
- systèmes d'exploitation répartis

Systèmes d'exploitation réseau

Ce type de systèmes d'exploitation gère généralement de façon non transparente les accès au réseau. Ce sont généralement des systèmes d'exploitation centralisés auxquels on ajoute des fonctionnalités et des outils pour accéder aux ressources distances et aux différents services sur le réseau tels que :

- Connexion à distance : rlogin, telnet, ftp, ssh
- Accès à des fichiers : sftp, FTPS, UUCP, ...

Cependant, de plus en plus de ces systèmes intègrent déjà des services permettant l'accès aux ressources distantes de façon transparente ou presque. On retrouve en particulier ce type de facilités au niveau de la gestion des fichiers avec des outils tels que NFS, Samba, OpenStack, ...

Systèmes d'exploitation répartis (DOS)

Les systèmes d'exploitation répartis (ou distribués) sont conçus à la base pour fonctionner avec les réseaux et ils les gèrent de façon transparente si cela est requis (localisation, migration, réplication, concurrence). Ces systèmes ont la particularité d'offrir une interface homogène pour tous. Concevoir de tels systèmes s'avère complexe car de nombreux attributs qui leur sont associés posent des problèmes d'implantation :

- performance
- capacité d'expansion (scalability)
- migration
- synchronisation (processus, horloge, ...)
- réplication
- fichiers
- fiabilité et tolérance aux fautes
- sécurité
- transparence (localisation, accès, panne, réplication, persistance, migration, relocalisation et transaction)

Les principaux concepts ou outils très utiles pour fournir ces attributs sont :

- Processus ;
- Fils d'exécution (thread) ;
- Clients/serveurs + RPC ;
- Messages ;
- Logiciels médiateurs ou intergiciels (middleware).

  Les logiciels médiateurs ou intergiciels (middleware) sont des environnements qui implantent un système «pseudo» réparti au dessus d'un système réseau. MPI, PVM, JINI (java), Corba, MoM, Web, Hadoop et OpenStack en sont quelques exemples.

Et voici quelques exemples de systèmes qu'on qualifie de «presque» distribués :

- NOW (Network of Workstation) : Boinc (Seti@Home,...), ...
- Grappe de calcul (Cluster) : MPI, ...
- Grid
- Cloud
  - Hadoop (HDFS, Map/Reduce, Distributed storage processing)
  - Openstack (Systèmes de fichiers Swift et Cinder, Nova - gestion des ressources de calcul, ...)
- P2P
- Internet

Logiciels médiateurs/intergiciels (Middleware)

Même si les réseaux sont omniprésents, il n'existe pas de réels systèmes distribués. Un compromis est donc requis pour faciliter l'intégration et la collaboration des ordinateurs sur un réseau et il se présente sous la forme de logiciels médiateurs (middleware).

Comme l'illustre la figure 7.7, un logiciel médiateur (middleware) est un environnement (souvent une bibliothèque) installé entre les systèmes d'exploitation de type réseau et l'application. Son rôle consiste à cacher les caractéristiques hétérogènes des systèmes sous-jacents et à faciliter les échanges, donc à fournir une plateforme qui assure aux applications la transportabilité, la transparence pour l'accès au réseau et l'inter-opérabilité de façon cohérente. Cette approche offre une combinaison acceptable qui ressemble dans bien des cas à un système distribué.

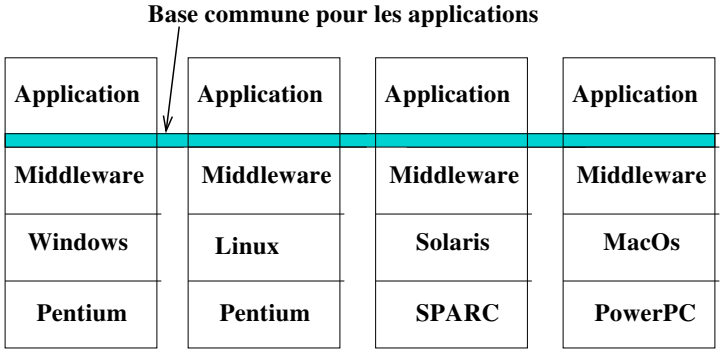

Figure 7.7 – Architecture des systèmes utilisant un logiciel médiateur (middleware).

Plusieurs types de logiciels médiateurs existent qui permettent d'accéder au réseau, notamment ceux :

- Basé sur les documents (Web) ;
  Ce type de système emploie l'environnement implanté par le Web, soit l'adressage par URL et le protocole HTTPS, qui ont l'avantage d'être uniformes pour tous.
  Dans ce cas, le applications sont conçues «au-dessus» du Web.
- Basés sur les fichiers ;
  Ces systèmes de fichiers tentent de dissimuler les frontières entre les ordinateurs (système de fichiers distribués). Des exemples de tels fichiers sont Samba, NFS, HDFS, GoogleFS et Swift.
- Basés sur les objets (CORBA) ;
  Ce type d'intergiciel offre un API pour développer des applications distribuées. En particulier, Corba fournit un langage de définition d'interface et un protocole inter-orb. Pour bien fonctionner, cet environnement exige la présence d'un serveur sur chaque plateforme appelé «ORB request Broker».
- Basés sur la coordination ;
  Des exemples de tels systèmes sont Linda et Jini.
- Basés sur le calcul : MPI, PVM ;
- Qui intègrent tout : Hadoop, Openstack.

7.2.3 Conclusion

Peu importe le type d'environnement choisi, implanter ou adapter un logiciel afin qu'il fonctionne adéquatement dans cet environnement présente son lot de défis. En particulier, aux niveaux des différents services offerts par un système d'exploitation plusieurs adaptations sont nécessaires pour fournir un accès aux ressources distantes, et c'est lors de ces adaptations que de multiples problèmes surgissent !

Nous allons donc présenter les différentes façons d'adapter tous les services d'un système d'exploitation aux systèmes multi-processeur et multi-ordinateurs. Rappelons que les différents services/fonctions d'un système d'exploitation sont :

- gestion de périphériques
- gestion de fichiers
- gestion de processus
- gestion de la mémoire
- ...

7.3 Systèmes de fichiers

Le système de fichiers est un composant clé de tout environnement. C'est l'un des premiers à avoir été modifié pour fonctionner sur des multi-ordinateurs, que l'on pense à NFS et Samba, ou même à des systèmes tels que Dropbox, Google Drive et OneDrive.

Les premières approches d'intégration au réseau ont consisté à permettre de transférer «explicitement» des fichiers d'un ordinateur à l'autre. Des protocoles tels que `FTP`, `UUCP` et `FTAM` ont alors vu le jour. Pour des besoins de sécurité, ces protocoles sont devenus `FTPS` ou `SFTP`. Plusieurs outils tels `wftp` et `Filezilla` les utilisent toujours.

Toutefois, la possibilité de transférer des fichiers s'avérait insuffisante pour partager ces ressources. Des systèmes de fichiers distribués ont donc été créés. En fait l'objectif est d'offrir le plus de transparence possible lors de l'accès aux fichiers, qu'ils soient locaux ou éloignés, du moins du point de vue de l'utilisateur.

Sous bien des aspects, ces systèmes ressemblent aux systèmes de fichiers centralisés. Ils comprennent :

- Le service de fichiers :
  C'est la spécification de tous les services offerts par le serveur au client (interface, API). Comme tout API cela comprend les fonctions primitives disponibles ainsi que les paramètres à utiliser. Cet API doit inclure les mêmes opérations qu'un système centralisé, soit principalement open(), close(), read(), write() et seek(), et celles-ci doivent s'adapter de façon transparente ou non à l'environnement distribué. Par exemple, pour l'ouverture d'un fichier, les syntaxes possibles pour ce service pourraient être :
  - accès à un fichier local : `open("fichier1")` ;
  - accès non transparent à un fichier distant :
    `open("132.210.40.88:/usr/local/foo/fichier1")` ;
  - accès transparent à un fichier distant : `open("/home/public/cours/ift630")`.

  Il y a deux concepts importants à distinguer dans les services de fichiers : les fichiers eux-mêmes et les répertoires.

- Le serveur de fichiers :
  C'est un processus qui implante les services ou du moins une partie d'entre eux. Il est envisageable de configurer l'environnement avec plusieurs serveurs. Dans un système distribué, leur nombre et leur localisation doivent normalement être transparents. Par exemple, aucun utilisateur (ou très peu) ne sait combien de serveurs offrent les services de Dropbox, ni où ils sont localiser. En fait, ils sont potentiellement répartis sur plusieurs sites (centres de données). De plus, chacun des serveurs peut implanter des services différents. Par exemple, certains serveurs peuvent être dédiés à la gestion des répertoires et d'autres à la gestion des fichiers.

7.3.1 Interface et attributs

Pour adapter/implanter adéquatement un système de fichier distribué, il est important de bien comprendre les différents concepts/objets qu'il implante ou manipule. Attardons nous à quelques éléments de réflexions :

- Qu'est-ce qu'un fichier ?
  La réponse à cette question affectera l'interface, en particulier les fonctions d'accès. De façon générale, un fichier, pour le système d'exploitation Unix, est une suite d'octets à laquelle le système n'accorde aucune signification particulière. C'est aux applications que revient la responsabilité d'interpréter ou de donner une signification aux octets contenus dans le fichier. De même, Windows n'interprète pas le contenu des fichiers.
  En fait, Unix et Windows attribuent un sens particulier à certains fichiers comme aux fichiers exécutables, non pas dans le but d'interpréter le contenu mais bien pour faciliter leur manipulation.
  Historiquement, certains systèmes ont tenté d'interpréter le contenu des fichiers.
- Attributs d'un fichier ?
  Les attributs standards d'un fichier (sur un système centralisé) sont : son nom, l'identification de son propriétaire, sa dimension, sa date de création, la date de sa dernière modification, les permissions, ...
  Ces mêmes attributs pour un fichier sont aussi nécessaires sur un système réparti en plus de quelques autres qui parfois s'ajoutent pour faciliter leur gestion. Par exemple :
  - Modifiable ;
    Un nouvel attribut important en serait un relié à la capacité de modifier ou non un fichier.
    Sur un système distribué, il est plus fréquent que plusieurs entités accèdent simultanément à un même fichier. Pour remédier aux problèmes de modifications simultanées, certains systèmes limitent les capacités qu'ont les utilisateurs à modifier le fichier. Les différents limitations sont :
    * fichiers immuables ;
      Pour régler les problèmes d'accès, certains systèmes rendent carrément les fichiers immuables, i.e. une fois le fichier créé, il est impossible de le modifier. Les opérations que ces systèmes fournissent sont : création, destruction et lecture (pas d'écriture).
      Historiquement, le système distribué Amoeba fut l'un des premiers à implanter des fichiers immuables. Toutefois, certains systèmes modernes offrent aussi des fichiers immuables, notamment pour gérer plutôt des versions d'un même fichier. C'est le cas de Git par exemple (qui est en fait un système de fichiers partiellement distribué mais sans transparence).

    * Ajout uniquement à la fin du fichier (append).
      Avec ce type de fichier, les modifications ne se font que par des ajouts à la fin du fichier. Il est impossible de modifier le contenu déjà existant. Le système de fichier de Google, `GoogleFS`, implante ce type d'attributs.

– Protection d'un fichier ?
  Comme cela a déjà été abordé dans le cours IFT320, la protection des fichiers est modélisée par une matrice d'accès et généralement implantée par des pouvoirs ou des listes d'accès.
  Pour rappel, les pouvoirs sont associés aux utilisateurs, Ils contiennent toutes les informations nécessaires pour donner à l'utilisateur qui le possède les accès aux ressources. Dans la vie, un billet de spectacle et l'argent sont des formes de pouvoir. Les listes d'accès, quant à elles, sont associées aux ressources. Quand un utilisateur désire accéder à une ressource, il doit figurer sur la liste pour obtenir (ou non) un accès. Dans la vie, la liste des réservations au restaurant, une liste de membres dans une assemblée syndicale ou la liste des votants dans une élection, constituent des listes d'accès.
  Sur les systèmes centralisés, les listes d'accès sont généralement employées (ACL ou autres). Leur utilisation est aussi courante sur le WEB. Toutefois ces listes posent un problème important, soit la multiplication des codes d'accès et des mots de passe.
  Sur le Web, le protocole d'authentification AUTH permet de garder des connexions ouvertes. Pour y parvenir, le client reçoit un jeton qui ressemble à un pouvoir. Lors de sa prochaine visite, la présentation de ce jeton suffira à l'authentifier (à détailler).

7.3.2 Architecture

La figure 7.8 présente l'architecture générale d'un système de fichiers centralisé. On y retrouve un processus (P1), qui accède à un fichier, soit par l'interface normale du système d'exploitation (open, read, write, close), soit par l'interface pour les fichiers projetés en mémoire centrale (*memory mapped file*). Une fois l'appel reçu, le noyau du système le redirige vers le gestionnaire de fichiers qui fait appel aux périphériques locaux pour accéder aux données reliées aux fichiers (disque, clé USB, ...)

La figure 7.9 représente l'architecture générale d'un système de fichiers distribués. Dans ce cas, le gestionnaire de fichiers accède aux périphériques locaux si les données sont emmagasinées localement. Si les données sont éloignées, alors la demande transitera par le réseau. Dans ce cas, au moins trois architectures sont possibles pour implanter le système de fichiers distribué :

- les disques distants (remote disk)
- le serveur de fichiers éloignés (accès éloigné)
- le serveur de fichiers éloignés (upload/download)

Les disques distants (remote disk)

La figure 7.9 présente l'architecture d'un système de fichiers distribués basé sur les disques distants. C'est une architecture client/serveur. Le client implante tout le système de fichiers incluant les algorithmes d'accès, c'est en fait le gestionnaire de fichiers. Il est aussi responsable de maintenir toutes les informations sur les fichiers ouverts (tables). Lors de l'accès à un fichier distant, le client retrouve les données par l'intermédiaire d'un pilote de disque virtuel qui fournit le même API qu'un pilote de disque local. Ce pilote virtuel communique avec un serveur sur le réseau pour accéder aux

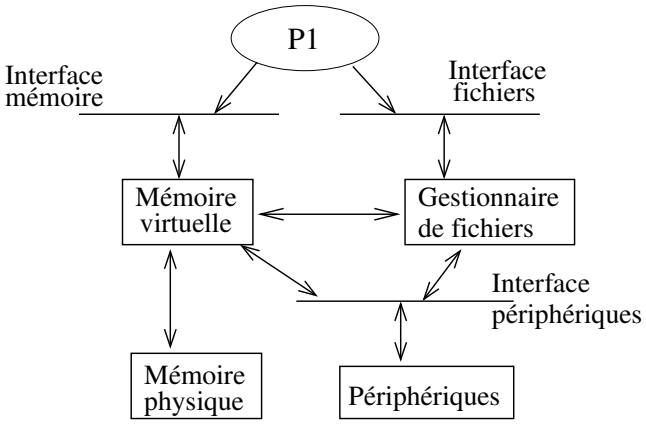

Figure 7.8 – Architecture d'un système de fichiers centralisé

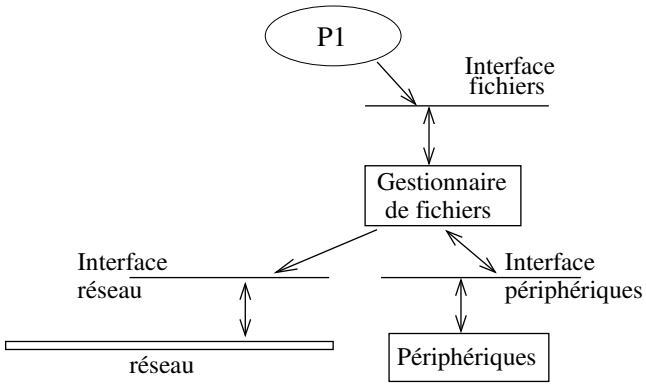

Figure 7.9 – Architecture d'un système de fichiers distribué basé sur les disques éloignés

périphériques distants. S'il n'y a pas de disque local, toutes les demandes d'accès aux fichiers sont dirigées vers le disque distant. Si un disque local est présent, le système de fichiers est chargé de déterminer si le fichier est emmagasiné localement ou non (table de montage).

Un serveur, localisé sur le réseau, fournit l'espace disque pour les clients et est responsable des accès aux disques distants. Ce serveur contient une application «disque distant» et un pilote virtuel. Le pilote reçoit les requêtes du réseau et les achemine à l'application. Cette dernière effectue les opérations sur le disque local et retourne le résultat par l'intermédiaire du pilote virtuel. Ce serveur fournit des commandes standards de type lecture, écriture et positionnement (*seek*) sur le disque.

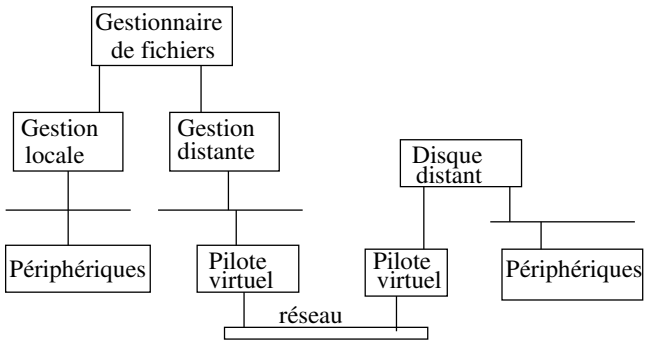

Figure 7.10 – Architecture d'un système de fichiers distribué basé sur les disques distants

Fait à remarquer, si la station ne possède pas de disque local, lors de l'exécution d'un programme, celui-ci est chargé à partir du disque distant. De même, la mémoire virtuelle se sert du disque distant pour la pagination et l'échange (*swap*).

Cette architecture était populaire dans les années 1980 avec les stations de travail sans espace disque (*diskless*). En effet, à l'époque les disques étaient trop coûteux pour être intégrés dans des stations de travail à priori peu coûteuses. Avec la baisse du coût des disques, vers la fin des années 1980, ce type de station de travail a disparu ainsi que cette architecture de système de fichiers. Toutefois dans les années 1990, cette architecture est réapparue avec le concept de terminal X (une station de travail minimaliste qui se connecte sur un serveur à distance).

L'avantage de cette architecture est qu'elle est simple. De plus, comme la gestion des accès aux fichiers est centralisée sur le client, des optimisations sont possibles pour améliorer les performances d'accès aux fichiers et minimiser la charge sur le réseau.

Cette approche assure aussi une bonne fiabilité. En effet, le serveur étant «stateless», un panne de ce dernier est facilement récupérable. Ainsi, comme il n'y a aucune information d'états conservée sur le serveur, la reprise après la panne est simple puisqu'aucune information n'est à reconstruire. Le serveur, une fois redémarré, n'a qu'à attendre les nouvelles demandes des clients.

Du côté des clients, lorsque le serveur tombe en panne, certaines de leurs demandes ne seront pas traitées. Comme le client ignore à quelle étape en était sa demande lors de la panne, il ré-exécute tout simplement les commandes (lecture, écriture, positionnement, état, ...). Comme les opérations de lecture et d'écriture sont idempotentes [131], il n'y a aucun problème à les répéter. Cependant, l'opération de positionnement de la tête de lecture («*seek*») relative à la position actuelle (ex. : seek +1) est une opération non idempotente. Le recours aux instructions de positionnement avec

des déplacements absolus, qui elles sont idempotentes, est donc nécessaire.

Idempotence

Une opération est dite idempotente [131] si une ou des application répétitives de celle-ci produira toujours le même résultat.
Par exemple, la valeur absolue est idempotente car :

$$abs(abs(abs(-5))) = abs(abs(-5)) = abs(-5) = 5.$$

Serveur distant

Selon l'architecture dite «serveur distant», l'une des parties du serveur de fichiers est locale et l'autre, distante. Le gestionnaire de fichiers est donc séparé en deux composantes. Les fonctions de chacune d'elles dépendent du modèle utilisé. Décrivons les deux modèles courants :

1. Le modèle basé sur le téléchargement/téléversement des fichiers (upload/download)
   Ce premier modèle, aussi appelé «file level caching», fontionne de la façon suivante :
   - lors de l'ouverture, le fichier est transféré en entier vers le client. le fichier est emmagasiné sur un disque local ou en mémoire centrale ;
   - les lectures/écritures se font localement ;
   - lors de la fermeture, le fichier est transféré en entier vers le serveur.

   Ce modèle a l'avantage que le serveur est simple et efficace. En effet, l'interface du serveur ne fournit que des commandes de téléchargement et de téléversement. Il est aussi efficace car la lecture/écriture sur disque peut être optimisée. Le transfert du fichier sur le réseau est aussi très efficace étant donné qu'il l'est en entier.
   Les inconvénients de ce modèle sont cependant nombreux. Premièrement, cette approche est coûteuse en espace et en bande passante. En effet, le client doit avoir suffisamment d'espace localement pour emmagasiner le fichier en entier. De plus, transférer le fichier en entier s'avère souvent inutile si seulement une fraction du fichier est effectivement utilisée. Cela entraîne une surcharge de la bande passante. De plus, comme plusieurs clients sont en mesure de transférer le même fichier, des problèmes de cohérence sont à prévoir. Cette situation est identique à celle qui survient avec la gestion de la mémoire cache. Les mêmes solutions sont applicables.
2. Le modèle basé sur l'accès à distance.
   Selon ce modèle, appelé «block caching», toutes les commandes sont transférées au serveur pour y être exécutées. Celui-ci fournit donc toutes les commandes standards d'un système de fichiers telles que ouverture, fermeture, lecture, écriture et déplacement. Ce modèle ressemble à l'architecture disque distant.
   Les avantages de ce modèle est de monopoliser moins de bande passante et moins d'espace sur le client. Autre avantage, en conservant des informations sur les fichiers ouverts, le serveur contribue à réduire le trafic sur le réseau. Certes, cela permet d'améliorer la performance mais le serveur et le client doivent alors maintenir de l'information sur les fichiers ouverts, ce qui entraîne de la duplication d'information et de potentiels problèmes de cohérence. Les autres inconvénients sont que le serveur, implantant plus de fonctionnalités, est beaucoup plus complexe que dans le modèle précédant et les accès aux fichiers beaucoup plus lents. Les optimisations s'avèrent aussi plus difficiles, du moins, pour l'usage de la bande passante.

Étant donné ces avantages et inconvénients, il est souvent proposé de combiner les deux approches en transférant une partie des fichiers dans une mémoire cache sur le client. Cela améliore potentiellement la performance mais on conserve les inconvénients reliés à la duplication d'informations et aux problèmes de cohérence.

Voici des exemples de systèmes de fichiers implantant (du moins partiellement) un de ces modèles :

- Dropbox
  Dropbox ne permet pas de modifier les fichiers sur place. Pour ce faire, il faut les télécharger, les modifier localement, puis les téléverser. Cette caractéristique le rapproche du modèle par téléchargement/téléversement.
- Google Drive
  Les fichiers de Google Drive sont modifiables sur place (sur le serveur). Cela ressemble plus à de l'accès à distance.
- CIFS (Samba, smb)
  Ce système de fichier est intégré à Windows, Linux et MAcOS. Il implante l'accès à distance, toutes les commandes sont transférées au serveur.
- NFS
  Ce système de fichier est principalement intégré à Linux. Il implante l'accès à distance, toutes les commandes sont transférées au serveur.
- HDFS (Hadoop)
  À venir !
- GoogleFS
  À venir !
- OneDrive
  Similaire à Google Drive avec l'intégration de Office365.
- owFS
  À venir !
- Manila (Cinder - OpenStack)
  À venir !

7.3.3 Service de répertoire

Pour les systèmes de fichiers, il existe un mode d'identification à deux niveaux (i.e. chaque fichier possède deux noms) :

- nom symbolique
  Cet identificateur est destiné aux utilisateurs du système (pour les usagers).
- nom binaire
  Cet identificateur est réservé à un usage interne du système. Les utilisateurs ne devraient jamais y avoir accès.

Le rôle du service de répertoire, implanté par le serveur de répertoire, est de faire le lien entre les deux. Ce service fournit en général :

- un alphabet et une syntaxe pour les noms symboliques ;
  En général les noms symboliques comportent des lettres de l'alphabet, des chiffres et quelques symboles. Ils ont aussi une longueur maximale.
- une hiérarchie de répertoire (arbre, graphe) ;

Afin de faciliter le classement et la recherche des fichiers, une vue hiérarchique des répertoires est fournie. Ainsi un répertoire peut contenir des sous-répertoires qui peut à son tour en contenir d'autres. Cette hiérarchie prend généralement la forme d'un arbre qui devient un graphe si les liens sont autorisés.
La hiérarchie en graphe pose certaines difficultés sur les systèmes centralisés lors de la destruction d'un fichier. Il y a un risque de créer des fichiers orphelins. Ce problème devient encore plus critique sur les systèmes de fichiers distribués car le graphe s'étend parfois sur plusieurs sites. Il devient donc plus ardus de localiser les orphelins. En effet, un ramasse miette exige d'arrêter toutes les activités du système pour fonctionner de façon fiable. Sur une site unique cela est possible. Sur plusieurs sites, cela devient extrêmement complexe.

- des liens symboliques ou physiques.
  Des liens sont des pointeurs vers des fichiers existants. Les liens symboliques prennent souvent la forme d'une entrée contenant le nom symbolique du fichier visé. Le lien physique contient le nom binaire du fichier.

Noms symboliques

Le concept de nom symbolique est relativement standard sur un système de fichiers centralisé. Dans une hiérarchie en arbre ou en graphe, il est composé de plusieurs noms composant le chemin vers le fichier (`/home/public/cours/ift630`).

Toutefois, pour sa contrepartie distribuée, plusieurs questions supplémentaires sur l'organisation et la gestion des noms symboliques se posent :

1. Quel est la perception de la hiérarchie ?
   Un élément clé de la conception d'un système de fichiers distribués est de déterminer si oui ou non, tous les sites «verront» exactement la même hiérarchie de répertoires. Une même «vue» signifie qu'un chemin est valide sur tous les sites. Cela est utile pour certaines applications telles celles utilisant MPI. Le système expérimental Amoeba fournissait une telle vue. Il est possible de fournir une telle vue sur les systèmes en configurant correctement les systèmes de montage (la commande «`mount`» de Unix).
   Une vue différente signifie que le comportement d'une application peut différer d'un site à l'autre. Ce type de vue s'implante aisément dans plusieurs systèmes en «montant» les systèmes à des endroits différents de la hiérarchie.
2. Existe-t-il une racine globale ?
   La plupart des systèmes de fichiers ont une racine distincte par site. Avoir accès à une racine globale pour tout le réseau permettrait de circuler librement dans tout le système de fichiers à la grandeur du réseau. Le système d'exploitation expérimental Amoeba était probablement le seul à fournir une racine globale.
3. Y-a-il un lien entre localisation et les noms ?
   Un autre élément important à considérer avec la gestion des noms est celui de déterminer si les noms garantissent une transparence de localisation et une indépendance de localisation.
   Fournir une transparence de localisation signifie que le nom du fichier n'indique pas où le fichier est emmagasiné. Un nom composé de l'identificateur de la machine suivi du nom de chemin local («*/id_machine/chemin_local....*») ne respecte pas ce principe. En effet comme l'identificateur du site fait partie du nom, il n'y a aucune transparence. En «montant» un système de fichier dans la hiérarchie locale, il est possible d'obtenir la transparence, si tout est bien configuré.

Fournir une indépendance de localisation signifie qu'il est possible de changer un fichier de place sans changer son nom. Il est possible d'obtenir une telle indépendance en configurant correctement le montage du fichier dans la hiérarchie locale. Une des techniques utilisée pour parvenir à cette indépendance est de gérer un espace de noms unique pour toutes les machines. À chaque fois qu'un fichier est créé, le nom est choisi dans cet espace. Le Web est organisé de cette façon. Tous les fichiers sur le Web ont un nom unique et peuvent changer d'adresse sans que leur nom en soit affecté.

Noms binaires

Comme nous l'avons déjà mentionné le service de répertoires fait le lien entre les noms symboliques et les noms binaires. Ainsi, lors de l'ouverture d'un fichier, le système recherche le nom binaire par l'intermédiaire du répertoire.

Mais qu'est-ce qu'un nom binaire ? La réponse à cette question est complexe car elle varie d'un système à l'autre. Sur les systèmes centralisés, c'est généralement un pointeur de type `I-node` ou autres.

Sur une système de fichier distribué, le nom binaire prend plusieurs formes mais il ressemblera généralement à un nom symbolique composé de l'adresse du serveur et du nom symbolique local. Ce «*nom binaire*» dans ce cas exigera une seconde traduction. Cela ressemble à un lien symbolique soit est une entrée de répertoire contenant le «nom du chemin» vers un fichier, celui-ci se situant potentiellement sur un autre site.

Sur un système de fichiers distribués, il est intéressant d'utiliser le concept de «pouvoir» comme nom binaire. Ainsi, ouvrir un fichier («`open(nom_symbolique)`»), nous donne un pouvoir (token) qui permet d'accéder directement au fichier distant. Ce pouvoir contient généralement l'identificateur du site (logique ou adresse sur le réseau) et le nom du fichier sur ce site. L'identificateur du site sert à envoyer un message vers le serveur détenant du fichier (qui effectuera la traduction du nom). Le serveur se chargera de l'accès au fichier. Si l'identificateur est une adresse logique, elle pourra être traduite en faisant appel à un serveur de nom que l'on rejoindra directement par une diffusion.

Un exemple de pouvoir est le lien que l'on obtient lorsque l'on partage des fichiers dans Dropbox, Google Drive ou Onedrive.

Il est possible pour un nom symbolique de posséder plusieurs noms binaires. Cela permet d'implanter la tolérances aux fautes en gérant plusieurs copies d'un fichier.

Sémantique de partage et performance

Afin de mettre en place un système de fichiers distribués rendant les mêmes services que sa contrepartie centralisée, il faut bien comprendre son fonctionnement, i.e. la syntaxe et la sémantique des fonctions spécifiées par l'API. La syntaxe est assez simple à reproduire de façon transparente ou non. Toutefois reproduire la sémantique de ces fonctions s'avère parfois très difficile, voire même impossible.

Explorons donc la sémantique de certaines fonctions d'un système de fichiers centralisés :

- Écriture
  Sur une système centralisé, une écriture prend effet immédiatement. S'il y a plusieurs écritures successives, celles-ci sont ordonnées selon le moment de leur exécution. Cette ordre s'appelle un **ordre total (total ordering)** ou un **ordonnancement absolu dans le temps (absolute time ordering)**. Cet ordonnancement est possible sur un système centralisé étant donné l'unique gestionnaire de fichiers se fiant à une horloge unique.

- Lecture
  Sur un système centralisé, une opération de lecture retourne la valeur de la dernière écriture. Cela est possible car toutes les écritures sont totalement ordonnées dans le temps.

Sur un système répartis, reproduire une telle sémantique devient complexe pour des raisons de performance et de par la présence de plusieurs horloges ainsi que des délais de communications. Explorons différentes situations :

- Un serveur unique
  Si le système de fichiers est géré par un seul et unique serveur, il est possible d'avoir un ordonnancement unique mais qui ne correspondra pas nécessairement à un ordonnancement total dans le temps ($\neq$ absolute time ordering). En effet, les délais de communication vers le serveur varient d'un client à l'autre faussant ainsi les temps d'arrivée sur le serveur. Ainsi une commande A émise par un client 1 peut n'arriver qu'ultérieurement au serveur par rapport à une commande émise par un client 2 même si celle-ci a été démarré avant (dans le temps).
  De plus, comme un seul serveur n'est pas suffisamment efficace, il est possible que les clients maintiennent un copie locale du fichier dans une mémoire «cache», particulièrement pour les fichiers fréquemment utilisés. Le client a donc le loisir de lire et de modifier directement sa copie du fichier. Cette situation risque d'entraîner des incohérences si le fichier original (ou une autre copie en cache) n'est pas mise à jour assez rapidement, comme le montre la figure 7.11.

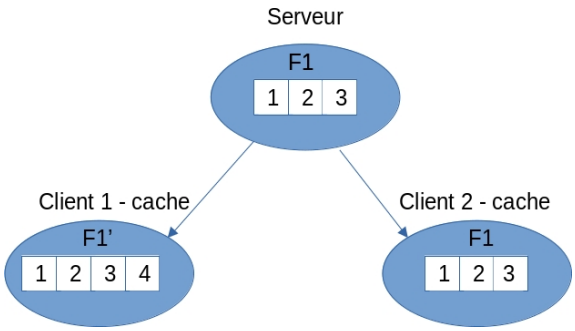

Figure 7.11 – Gestion d'une mémoire cache avec un seul serveur

Il existe plusieurs solutions pour éviter ou du moins minimiser l'impact des incohérences de données dues à l'usage d'une mémoire cache :

1. Propagation immédiate
   Cette solution consiste à propager immédiatement vers le serveur toutes les modifications effectuées au fichier. Cette solution ne résout pas entièrement le problème puisque les copies des autres clients demeurent inchangées, donc incohérentes. De plus, cette solution n'est pas efficace et risque de surcharger le serveur.
2. Sémantique de session
   Il est possible de relaxer la sémantique des opérations de lecture et d'écriture afin de rendre les opérations plus efficaces. Ainsi, au lieu d'exiger qu'une lecture «voit» les effets de toutes les précédentes écritures, on instaure une règle du type :

   > *Les modifications sont propagées vers le serveur seulement à la fermeture du fichier.*

Cette sémantique, appelée «sémantique de session», a l'avantage d'être beaucoup plus efficace et de ne pas surcharger le serveur. Toutefois, **que se passe-t-il lorsque deux processus modifient le même fichier simultanément (dans leur cache respective)** ? Ces modifications ne seront évidemment propagées qu'à la fermeture des fichiers par chacun des clients. Cette situation doit être considérée et résolue. Les solutions standards sont, soit de conserver seulement la dernière version du fichier fermé, soit de n'en conserver qu'une seule choisie au hasard. Dans les deux cas, des informations seront perdues. C'est le principal inconvénient de cette sémantique. Cependant elle est basée sur le fait que ce type d'événement est peu fréquent.

3. Fichier immuable
   Un autre approche consiste à rendre les fichiers immuables. Selon cette sémantique, il n'est plus possible de modifier un fichier, mais il est possible de le remplacer par un autre. Ainsi, toute modification crée un nouveau fichier. Autrement dit, pour modifier un fichier, il faut le lire, le mettre à jour en cache puis créer un nouveau fichier du même nom (détruisant ainsi l'ancien).
   Remarquons que même si le problème de mise à jour des fichiers disparaît, il est toujours possible de modifier (simultanément !) les répertoires. Il faut donc contrôler les créations simultanées de fichiers. Cette opération de création consiste à, soit choisir la dernière version sauvegardée, soit choisir une version au hasard, ou encore utiliser des transactions atomiques.

Une autre difficulté survient avec l'usage de la mémoire cache. En effet celle-ci rend caduc un autre élément de la sémantique relié aux fichiers. Ainsi, dans Unix, on associe à chaque fichier ouvert un pointeur vers la position courante dans le fichier. Les lectures et écritures se font à cette position.
Ce pointeur est normalement partagé entre le processus qui a ouvert ce fichier et ses enfants. Malheureusement, lorsque les enfants s'exécutent sur des machines différentes et recourent à une cache, le maintient de cette sémantique devient impossible. Chaque processus possède alors son propre pointeur.

- Plusieurs serveurs
  Un seul serveur n'est pas efficace sur un système réparti. En effet, comme il doit traiter toutes les demandes de tous les clients sur le réseau, il devient généralement assez rapidement un goulot d'étranglement. Il est donc courant de faire appel à de multiples serveurs. Dans cette situation, un ordre total n'est plus possible. Nous reviendrons sur ce sujet.

7.3.4 Implantation

Avant tout, quelques observations importantes en lien avec les systèmes de fichiers suite à certaines études :

- la plupart des fichiers sont petits ;
  Dans les années 80, la taille moyenne de la majorité des fichiers était inférieure à 10k. Même si celle-ci est probablement plus grande aujourd'hui, elle demeure relativement petite.
  L'implication de cette observation est qu'il est possible et facile de transférer le fichier en entier vers le client.
- la plupart des fichiers ont une courte durée de vie ;
  La durée de vie moyenne des fichiers est de quelques milli-secondes (fichiers temporaires). Il est donc possible de créer et de conserver le fichier dans la cache du client jusqu'à sa destruction

sans jamais le transférer vers le serveur.
- peu de fichiers sont partagés ;
  Les fichiers partagés constituent plutôt des exceptions que la règle. En effet, les fichiers situés dans les répertoires des utilisateurs sont rarement partagés. En conclusion, les mettre en cache ne présente guère de risques d'incohérence.
- différents types de fichiers impliquent différents traitements ;
  Certains fichiers (exécutables, bibliothèques, ...) sont rarement modifiés. Les répliquer s'avère alors (quasi) sans risque. Comme déjà mentionné, les fichiers temporaires peuvent être emmagasinés seulement chez le client.
  On remarque donc qu'en caractérisant les types de fichier, il est possible d'optimiser les opérations de notre système de fichiers.

Lorsqu'il s'agit de concevoir un système de fichiers distribués, il y a plusieurs décisions techniques à prendre quant à sa structure.

Structure : client vs serveur

Un élément important de la structure consiste à déterminer s'il y aura une distinction entre les clients et les serveurs. Aucune distinction signifie que les clients et les serveurs exécuteront le même logiciel. Un client pourrait donc aussi agir comme serveur. Les premières versions du système de fichier distribué NFS [12, 139, 140, 80] ne faisait aucune distinction entre un site client ou un site serveur. Tous les sites exécutent la même version du logiciel et offrent ainsi des services de fichiers. Il est donc concevable de «monter» des systèmes de fichiers à partir de tous les sites ayant installé NFS. Avec NFS, le système de fichiers distribués est intégré au noyau.

Avec d'autres systèmes, tels Amoeba [122, 105, 66] et Mach [133, 89], les serveurs des fichiers (et de répertoires) sont des processus comme les autres. Un environnement est donc susceptible d'exécuter seulement le serveur, seulement le client ou les deux.

Le système de fichiers distribué du système d'exploitation expérimental Amoeba [122, 105, 66] fait une distinction plus nette entre les clients et les serveurs. Comme mentionné précédemment, le serveur de fichiers (un processus) est distinct du client (un autre processus). Cependant, un site hébergeant un serveur est fondamentalement différent d'un site hébergeant seulement un client. En effet, le serveur est matériellement configuré différemment et exécute normalement une version différente du noyau du système d'exploitation. Ce noyau est optimisé pour exécuter les serveurs de fichier (gestion cache, ...)

Structure : fichier vs répertoire

Autre élément de conception à considérer, la distinction claire ou non entre fichiers et répertoires. Un unique serveur doit-il être responsable de gérer les fichiers et les répertoires ou utilisera-t-on des serveurs distincts ?

S'il n'y a qu'un seul serveur, celui-ci implante toutes les fonctions tels que `open`, `close`, `read` et `write`.

S'il y a deux serveurs, la fonction `open` fera appel au serveur de répertoire qui lui, retournera le nom binaire du fichier. Les appels suivants de lecture et d'écriture feront directement appel au serveur de fichiers.

Cette séparation possède quelques avantages :

- Émuler plusieurs structures de répertoires en utilisant différents serveurs de répertoires pour un même serveur de fichiers. Par exemple, il serait possible d'émuler la VFat, Unix ou NTFS par le biais de trois serveurs de répertoires distincts.
- la conception est simplifiée par le fait que la structure est plus modulaire, chaque serveur étant plus limité en terme de services offerts.

Un inconvénient de cette approche est qu'elle requiert plus de communications entre les différents serveurs.

Notons que les systèmes de fichiers de Linux et Windows n'emploient qu'un seul serveur. Le système expérimental Amoeba utilisait deux serveurs distincts. Les systèmes de fichiers GoogleFS et le Web quant à eux, font appel à des serveurs distincts.

Structure : recherche d'un nom

Une autre décision de conception reliée à la structure est en lien avec la méthode implantée pour la recherche d'un nom de fichier s'il y a plusieurs serveurs ?

Plusieurs choix sont possibles pour cette recherche :

1. **par l'usager lui-même** qui se charge d'interroger chaque serveur tour à tour et ces derniers lui retournent soit le nom du ficher, soit le nom du prochain serveur pour poursuivre la recherche (voir la figure 7.12).
2. **par le serveur lui-même** qui, suite à une demande d'un client, se charge de la transférer au serveur suivant, s'il ne détient pas le fichier. Ce transfert se poursuit jusqu'à ce que le fichier soit trouvé (voir la figure 7.13).
3. **par une commande de diffusion** qui communiquera la demande à tous les serveurs simultanément. Seul le serveur détenant le fichier répondra positivement (voir la figure 7.14).

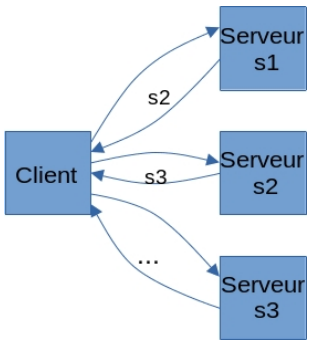

Figure 7.12 – Recherche par le client

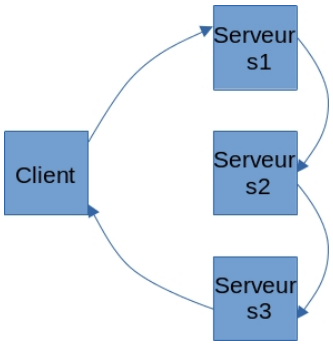

Figure 7.13 – Recherche par les serveurs

Structure : Stateless (NFS) ou statefull (AFS et RFS)

Un serveur est dit «**statefull**» **(avec états)** s'il maintient de l'information sur les fichiers ouverts par les clients et sur les clients eux-mêmes. Les systèmes de fichiers AFS, RFS, SMB sont tous «statefull».

En revanche, un serveur est dit «**stateless**» **(sans états)**, s'il ne maintient aucune information sur les fichiers ouverts par les clients. Dans ce cas, chaque demande ressemble à une transaction.

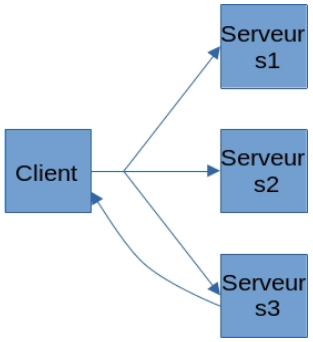

Figure 7.14 – Recherche par diffusion

Elle doit être complète, en ce sens qu'elle doit contenir toute l'information nécessaire pour effectuer le traitement sur le fichier. Le système de fichier NFS est un système «stateless».

Chacune de ces approches possède ses avantages et ses inconvénients. En voici un aperçu selon las facteurs suivants :

- taille des messages ;
  Avec un serveur **sans états**, les messages sont plus longs. En effet, ils doivent contenir plus d'informations telles que le nom du fichier, la position dans le fichier, ...
- tolérance aux pannes ;
  Un serveur **sans états** est plus tolérant aux pannes. En effet, comme il ne conserve aucune information sur les clients et leurs fichiers, la reprise après une panne est très simple. Il suffit de redémarrer et d'attendre les nouvelles demandes. Au contraire, un serveur **avec états** maintient des informations telles qu'une table des fichiers ouverts et les clients auxquels ces informations sont associées. Lors d'une reprise, il doit reconstruire cette table. Cela implique parfois de communiquer avec tous les clients pour retrouver toutes les données.
- Espace ;
  Comme un serveur **sans états** ne maintient aucune information sur les fichiers ouverts, il occupe donc moins d'espace en mémoire.
- performance ;
  Un serveur **avec états** est généralement plus performant car l'information qu'il conserve lui permet d'optimiser certaines requêtes. Ainsi, il lui est envisageable de faire des lectures à l'avance (*read ahead*), les messages sont plus courts, le verrouillage est possible et l'idempotence est plus facile si les communications ne sont pas fiables.
- verrouillage de fichier ;
  Dans certaines situations, l'action entreprise par un client requiert que l'on verrouille un fichier. Cette opération est simple à exécuter dans le cas d'un serveur **avec états**. Cependant, avec un serveur **sans états**, cela s'avère vraiment complexe. En effet, comme le serveur ne retient aucune information sur l'utilisation du fichier, celui-ci est difficilement capable de détecter un partage de fichier, de l'éviter ou bien de l'empêcher.
  Le système de fichiers NFS parvient toutefois à verrouiller un fichier en modifiant temporairement les autorisations d'accès.

Structure : gestion de la cache

Dans le modèle de gestion de fichiers basé sur des clients et, un ou plusieurs serveurs, il est nécessaire de faire appel au concept de mémoire cache afin de garantir une bonne performance. Cependant, la conception d'un système de gestion de cache implique maintes décisions qui influencent autant la performance que la sémantique du système visé.

Quelques choix de conception concernent particulièrement :

- **La localisation de la cache ;**
  Il y a en fait quatre endroits possibles pour implanter la mémoire cache du système de fichiers :
  1. **Sur le serveur** (disque ou mémoire centrale) (server caching) ;
     Une mémoire cache localisée sur le disque du serveur signifie en fait qu'il n'y a pas de cache. Nous n'en parlerons donc pas.
     Lorsque la mémoire cache est localisée en mémoire centrale du serveur, cela signifie que les fichiers récemment accédés sont conservés dans la mémoire centrale du serveur. Cela évite les transferts sur le disque mais n'améliore pas vraiment la performance. Cette approche s'implante toutefois facilement et ne présente aucun risque d'incohérence des données.
  2. **Sur le client** (disque ou mémoire centrale) ;
     Lorsque la mémoire cache est située chez le client, on élimine les accès au réseau. Le gain en performance est donc significatif. Cependant, encore une fois, la cache est localisée soit sur le disque du client, soit en mémoire centrale. Plusieurs systèmes localisent la cache sur le disque (NFS, AFS), ce qui est moins performant mais plus fiable en cas de panne.
     D'autres systèmes implantent la mémoire cache en mémoire centrale, mais ce choix implique d'autres décisions quant à la localisation de la cache. En fait, elle se situe soit dans la mémoire du processus usager, soit dans un espace mémoire dédié dans le noyau du système, soit dans la mémoire d'un processus indépendant, le gestionnaire de cache. La figure 7.15 présente ces trois possibilités.

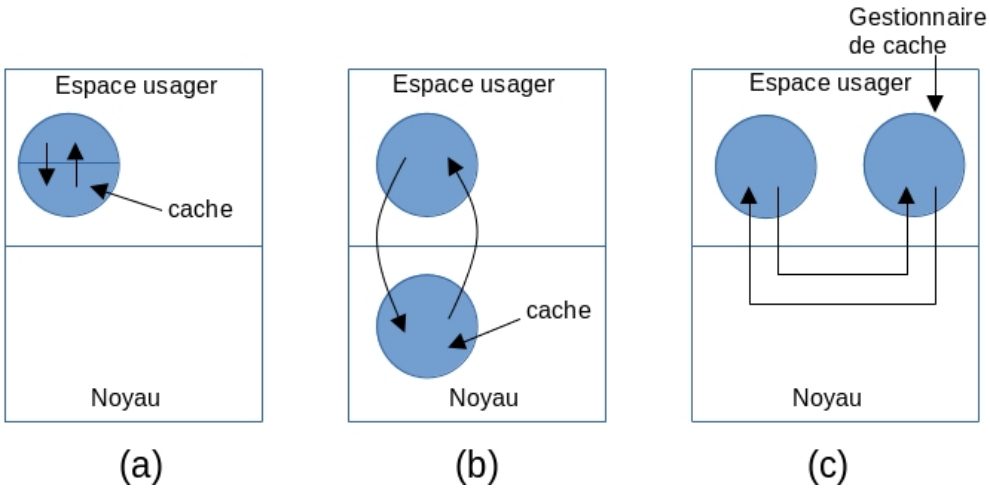

Figure 7.15 – Différentes possibilités de localisation de la cache en mémoire centrale.

Si la cache est localisée dans la mémoire du processus usager (a), l'accès est extrêmement rapide. Il n'y a guère de surcharge de travail pour l'environnement («overhead»). Lorsque le processus termine, les modifications sont retournées au serveur et la cache est détruite. Si la cache se situe au niveau du noyau, l'accès est un peu plus lent car il implique une certaine surcharge de travail pour le système (appels systèmes requis). Cependant, contrairement au cas précédant, une fois le fichier en cache, il devient disponible pour tous les processus. De plus, comme le contenu de la cache «survit» à la fin du processus initiateur, son contenu peut être ré-utilisé plus tard si cela s'avère nécessaire. Finalement, l'allocation dynamique d'espace dans le noyau est simple et plus efficace que dans l'espace usager.
Si la cache est localisée dans un processus spécialisé, soit le gestionnaire de cache, l'accès est encore plus lent car les échanges doivent se faire par des appels systèmes. Néanmoins, le code de gestion de la cache n'étant pas dans le noyau, il le simplifie (plus modulaire) et le rend plus facile à développer.

- **transfert entier ou partiel des fichiers ;**
  Généralement, la politique doit déterminer quels sont les fichiers ou parties de fichier à conserver dans la cache. Un choix possible est de toujours conserver des fichiers en entier. Cela implique de copier le fichier en cache dès son ouverture. Si on maintient seulement des parties de fichiers dans la cache, celles-ci sont chargées seulement au besoin. Cela permet d'occuper moins d'espace dans la cache.
- **la politique de gestion de la cache lorsqu'elle est pleine ;**
  Comme l'espace alloué à la cache n'est pas infini, il est possible que celle-ci soit pleine. Il est important de remarquer que cette situation est similaire à à celle de la gestion de la mémoire virtuelle lorsque la mémoire centrale est pleine. Les algorithmes requis sont donc les mêmes. L'algorithme le plus commun est LRU.
  Si les fichiers sont chargés en entier, ce cas ressemble au «swapping» sinon cela s'apparente à la pagination.
- **la gestion de la cohérence de la cache ;**
  La présence d'une mémoire cache risque fort de provoquer des incohérences dans les données. Ces cas risquent de se produire seulement lorsqu'au moins deux clients modifient le fichier en même temps. Parmi les solutions sont proposées, plusieurs modifient la sémantique des fichiers en se contentant de noter que des incohérences sont, jusqu'à un certain point, normales et acceptables. Dans certains cas, des modifications sont perdues et des actions doivent généralement être entreprises pour les récupérer. Les différentes solutions proposées pour gérer la cohérence de la cache sont :
  1. **Ne pas en tenir compte ;**
  2. **Écriture directe (write through) ;**
     Selon cette approche, aussitôt qu'une modification est faite, elle est transmise au serveur. Elle est donc rapidement rendue visible à tous les utilisateurs. Certes, cela n'empêche pas entièrement les incohérences. En effet, si un fichier est conservé en cache sur une longue période de temps, il est possible que celui-ci ne soit pas à jour selon les dernières modifications apportées à la copie originale. Pour éviter les incohérences, le gestionnaire de cache doit interroger régulièrement le serveur pour s'assurer de la validité de sa copie (date, version, ...). Il serait possible de concevoir un système dans lequel le serveur invalide les caches quand des modifications sont apportées à l'originale, mais cela l'obligerait à maintenir une liste de tous les clients, ce qui s'avère généralement complexe (s'il y a

beaucoup de clients).
Cette approche élimine le trafic des lectures mais pas celui induit par les écritures. Le gain de performance n'est donc pas optimal.

3. **Écriture avec délai ;**
Selon cette politique, on patiente soit un certain temps pré-défini, soit qu'un certain nombre de modifications soient réalisées avant de les transmettre en bloc au serveur. Évidemment, selon cette approche, il est encore possible de perdre de l'information.
Cette approche s'apparente à la sauvegarde automatique des éditeurs de fichiers. En effet, ceux-ci sont souvent configurés pour sauver le fichier suite à un nombre fixe de modifications. Cela évite une perte totale en cas de panne.
Le système de fichier NFS implante cette politique. Il transmet les mises à jour au serveur à toutes les 30 secondes par défaut.
4. **Aucune transmission ;**
Ne jamais transmettre les modifications au serveur est aussi un choix. Cela s'applique aux fichiers temporaires dont la durée de vie est très courte. Ces fichiers sont donc créés et conservés en cache en permanence. Le serveur n'est même pas conscient de leur existence.
5. **Sémantique de session ;** (*write on close*)
Comme déjà présenté, avec cette politique les mises à jour sont transmises au serveur seulement à la fermeture du fichier. Dans certains cas, il est possible de mettre un délai supplémentaire sur la transmission au cas où les mises à jour seraient ré-utilisées ou détruites.
6. **Contrôle centralisé ;**
Selon cette politique, le serveur surveille toutes les activités effectuées sur les fichiers. Il est donc au courant de toutes les opérations exécutées et les contrôle, en particulier les demandes d'ouverture. Il est donc capable, lors d'une ouverture en mode écriture, d'agir sur les caches des clients.
L'effet négatif probable est la surcharge du serveur.
7. **Invalidation de cache ;**
Selon cette politique, aussitôt qu'une modification sur un fichier particulier est rapportée au serveur, ce dernier invalide toutes les caches des clients concernant ce fichier. Un message doit être expédié à tous les clients leur demandant de détruire la copie locale du fichier. Lors des prochaines opérations, la nouvelle version du fichier sera rechargée dans la cache.
Cette politique est courante au niveau des caches matérielles et fonctionne très bien car le nombre de cache est fixe. Toutefois au niveau du serveur logiciel, elle pose un certain nombre de difficultés. Ainsi, pour un serveur, les clients sont généralement anonymes et il ne maintient pas de liste de clients. De plus, maintenir une telle liste s'avère complexe si le nombre de clients est très grand et varie constamment. Enfin, un serveur devrait envoyer des réponses aux clients (suite à des requêtes) plutôt que des demandes.
8. **Élimination de la cache lors d'une écriture ;**
Cette façon de faire constitue un cas particulier de la précédente. Ainsi, si le serveur détecte que plusieurs clients demandent le même fichier en lecture et en écriture, il a l'option d'éliminer la mise en cache. Il envoie donc un message à tous les clients pour les obliger à détruire la copie du fichier dans leur cache locale. À partir de ce moment, toutes les opérations se feront à distance.
Cette approche pose évidemment les mêmes problèmes que l'approche précédente.

Structure : gestion des copies multiples (réplication)

Lors de la conception d'un système de fichiers, on doit déterminer si les copies multiples d'un même fichier sont supportées. La réplication apporte plusieurs avantages :

1. **meilleure fiabilité ;**
   La réplication améliore la fiabilité du système. En effet si une copie est perdue ou corrompue, une autre copie prend la relève. C'est une forme de copie de sécurité (backup).
2. **meilleure disponibilité ;**
   En cas de panne d'un serveur ou d'un disque, les fichiers demeurent disponibles (en supposant que les copies sont localisées sur un autre serveur ou disque).
3. **meilleure distribution de la charge de travail ;**
   Si les multiples copies des fichiers sont réparties sur plusieurs serveurs, la charge de travail se répartit également sur différents serveurs.
4. **meilleur temps d'accès.**
   Détenir de nombreuses copies d'un même fichier améliore le temps d'accès de plusieurs façons. La probabilité qu'une des copies soit plus accessible à proximité est augmentée de même que la possibilité de choisir un serveur moins surchargé de travail pour répondre à la demande.

Même si elle présente de réels avantages, la réplication pose toutefois plusieurs défis lors de la conception et de l'implantation. Tout d'abord, on doit déterminer si la création des copies d'un fichier est transparente ou non. Si elle n'est pas transparente, elle est contrôlée par l'usager (du moins partiellement). Comme l'indique la figure 7.16, le client doit se charger de communiquer avec tous les serveurs qui hébergeront les futures copies.

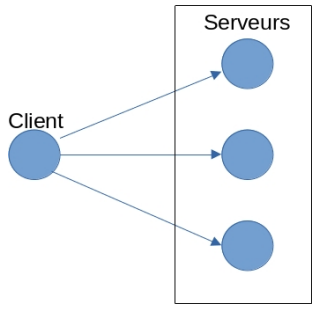

Figure 7.16 – Gestion non transparente de la réplication

Si la création des copies est transparente pour le client, le système se charge de cette gestion. La figure 7.17 présente deux techniques usuelles pour assurer ce travail, soit la réplication paresseuse et la communication de groupe. Selon la réplication paresseuse (figure 7.17 (a)), le système crée une copie principale et ultérieurement, lorsque le serveur sera disponible (ou sous une charge légère), il créera de nouvelles copies. Avec la communication de groupe, toutes les requêtes sont diffusées à tous les serveurs du groupe, créant ainsi automatiquement toutes les copies. Il est important de noter que la communication de groupe [20, 52] assure généralement l'ordonnancement des messages.

Il existe d'autres modes de gestion pour la création des copies multiples que nous n'aborderons pas ici.

Une fois les copies créées, il est alors important de gérer leurs mises à jour. La politique de mise à jour est cruciale pour éviter l'introduction d'incohérences entre les copies. Voici quelques

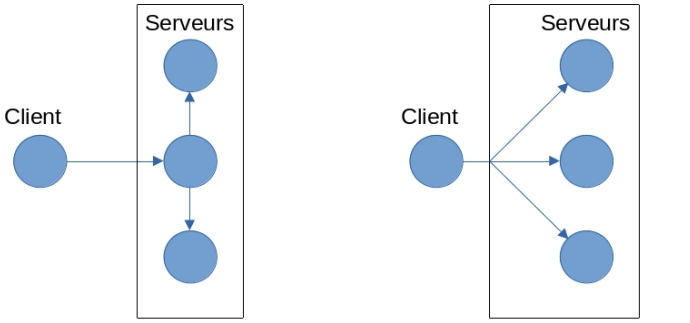

Figure 7.17 – Gestion transparente de la réplication

approches pour gérer les mises à jour :

1. **un message à chaque copie ;**
   Selon cette approche, lors d'une modification à un fichier, un message contenant les données altérées est envoyé à chaque copie du fichier. Cette façon de procéder ne fonctionne pas bien car elle risque de provoquer des incohérences irrécupérables. En effet, il est possible que les messages, expédiés par plusieurs clients, n'arrivent pas dans le même ordre aux différents serveurs ou ne se rendent tout simplement pas. Quand ils sont reçus dans le désordre, plusieurs versions du même fichier seront créées. Si un message est perdu, certaines modifications ne seront pas appliquées et, encore une fois, plusieurs versions seront créées (certaines avec la mise à jour et d'autres sans).
   Pour corriger ces problèmes, il est possible d'utiliser des fonctions de diffusion atomique (atomic broadcast [126]) ou de diffusion causale (causal broadcast) [20].
2. **copie primaire (primary copy replication) ;**
   Selon cette approche, un serveur est considéré comme le principal récepteur des mises à jour. Il les reçoit toutes et envoie les modifications aux serveurs secondaires.
   Cette approche n'est pas très fiable car, si le primaire tombe en panne, tout le système de fichiers le devient aussi (du moins temporairement) et certaines mises à jour seront éventuellement perdues.
3. **vote ;**
   Dans le cas d'un vote, les lectures et écritures se voient associées un quota de vote. La somme des quotas de lecture et d'écriture doit être plus grande que le nombre total de copies.
   Lors d'une écriture, la requête est expédiée à tous les serveurs. Si le nombre de réponses positives à la demande est plus grand ou égal au quota, la mise à jour est alors effectuée. On procède de la même façon pour les lectures.
4. **autres.**
   D'autres approches seront éventuellement présentées dans le cours systèmes répartis et sont traitées dans Tanenbaum [106, 107] et Colouris [20].

Fiabilité

Un système de fichiers distribué se doit d'être fiable. Cependant, il est fort possible, sur un réseau d'ordinateurs, que des commandes distantes se perdent ou du moins en apparence (horloge de garde). Quand cela se produit, il y a trois causes possibles :

1. **la commande ne s'est pas rendue ;**
   Si la commande ne s'est pas rendue, on peut la re-transmettre sans que cela ait de conséquences fâcheuses. Si jamais elle ré-apparaissait, les deux commandes seraient exécutées.
2. **le résultat (la réponse) s'est perdu ;**
   Dans ce cas, la commande a été exécutée par le serveur mais la réponse s'est perdue. Il est alors possible de re-transmettre l'opération si elle est idempotente [2]. Dans le cas contraire, une forme plus complexe de reprise devra être envisagée auprès du serveur .
3. **la réponse est retardée.**
   Si la réponse ne s'est pas réellement perdue est qu'elle réapparaît, on reçoit alors plusieurs réponses pour une même requête. Il faut donc prévoir une méthode pour éliminer les réponses redondantes.

Il est malheureusement impossible pour un client de détecter laquelle de ces trois situations s'est produite. Sa réaction sera donc la même dans les trois cas. Il initialise d'abord une horloge de garde et lorsqu'elle vient à échéance, si elle est idempotente, il répète l'opération. Par la suite, le client et le serveur doivent gérer les difficultés provoquées par l'apparition de commandes/réponses multiples (si une commande ou une réponse ré-apparaît). Que se passe-t-il pour chacune des opérations :

- **de lecture ?**
  L'opération de lecture est idempotente. Si elle semble perdue, elle est re-transmise autant de fois que nécessaire par le client et ré-exécuter par le serveur sans que cela ne fausse le résultat. Néanmoins, le client devra tenir compte des multiples réponses. Des numéros de version sur ces dernières permettront de détecter les doublons.
- **d'écriture ?**
  L'opération d'écriture est également idempotente. Sa ré-exécution par le serveur ne provoquera aucune incohérence. Toutefois le client devra encore une fois gérer les copies des réponses redondantes.
- **de positionnement ?**
  L'opération de positionnement dans un fichier n'est pas idempotente si elle est relative (du type «`seek + 1`»). Elle l'est seulement si elle est absolue. Le client doit adapter son opération en conséquence s'il doit la re-transmettre.

De plus, pour garantir la fiabilité des systèmes de fichiers, on utilise fréquemment la **mémoire stable** (stable storage), en particulier pour assurer l'atomicité des écritures. On a généralement recourt aux **disques RAID** pour implanter ce service. L'annexe A présente en détails ces concepts.

Exemples d'implantation

Au fil des années, plusieurs systèmes de fichiers distribués ont été conçus et implantés. Certains sont demeurés expérimentaux et d'autres sont couramment intégrés dans nos environnements modernes.

- NFS (Unix, Sun) [20, 106] ;
- AFS (Andrew File System) [20, 112, 123, 36, 9], OpenAFS (IBM) [70, 124] ;

2. Une opération est idempotente lorsque, même exécutée plusieurs fois, le résultat est toujours le même.

- NCP (Novell) [138, 35] ;
- Google FS [38, 39, 117, 130] ;
- HDFS (Hadoop) ;
- Swift [73, 74] et Manila [72] (OpenStack) ;
- S3 (Amazon) ;
- Azure (Microsoft) ;
- Samba (smb,) ;
- Locus, Coda, ...

7.4 Gestion de l'UCT

La gestion de l'UCT est la seconde fonctionnalité d'un système qui doit être adaptée pour supporter la présence de multiples processeurs/cœurs ou même de plusieurs ordinateurs (grappes, grid, ...). Dans ce contexte, on ne parle plus de gestion de l'UCT mais bien de gestion des UCTs. La travail du planificateur change donc légèrement.

Rappelons que les objectifs d'un planificateur, tel que vu dans les cours précédents, sont de minimiser les temps d'exécution des processus et d'assurer une bonne utilisation des ressources (principalement de l'UCT mais aussi de toutes les unités périphériques).

Avec l'apparition de multiples processeurs, les objectifs du planificateur sont légèrement modifiées. Minimiser le temps d'exécution demeure encore le but principal. Toutefois celui visant une utilisation optimale des ressources, notamment du processeur, est révisé. Il vise dorénavant une utilisation optimale et balancée de tous les processeurs. En particulier, le planificateur tentera toujours de balancer la charge de travail entre tous les processeurs.

Évidemment la gestion des systèmes multiprocesseurs ou multi-cœurs diffèrent de celle des systèmes multi-ordinateurs. La présence d'une mémoire commune dans le premier cas permet une gestion beaucoup plus efficace. L'absence de mémoire commune et la présence du réseau dans le second cas, forcent le recours à des solutions complètement différentes.

Dans les deux cas toutefois, il faut décider si l'assignation est statique ou dynamique. Une assignation statique associe un fil d'exécution ou un processus à un processeur pour toute le durée de son exécution. Une association dynamique permet à un fil de changer de processeur (migration) au cours de son exécution. La présence de mémoire commune dans le cas des multi-processeurs rend cette migration plus simple même si elle a un coût. Sur les multi-ordinateurs, cette migration s'avère beaucoup plus coûteuse et présente de grands défis.

Dans les sections qui suivent nous présentons les éléments à prendre en considération pour concevoir des algorithmes pour chacun de ces deux environnements. Nous donnons également quelques exemples d'algorithmes.

7.4.1 Multi-processeurs

Par multi-processeurs, nous signifions des systèmes supervisant des ordinateurs ayant plusieurs cœurs ou processeurs qui se partagent la mémoire centrale. Le système doit donc gérer tous les processeurs afin d'obtenir un maximum d'efficacité. Encore faut-il déterminer ce que veut dire «efficacité» dans ce contexte particulier.

Attardons-nous dans un premier temps à ce qui différencie une planification sur mono-processeur (telle que vue en IFT320) de celle sur multi-processeurs et à ce qui rend cette dernière beaucoup plus complexe.

Lorsque l'on conçoit un planificateur, il faut déterminer s'il gère des fils d'exécution ou des processus. Sur des systèmes ne reconnaissant pas les fils d'exécution, ou uniquement ceux de niveau usager, on planifie seulement les processus. L'ajout des fils d'exécution de niveau noyau rend la planification plus complexe et ce, à bien des points de vue.

Il faut d'abord décider ce que l'on planifie : les fils d'exécution, les processus ou les deux. Il est en effet possible de ne planifier que les fils sans se préoccuper des processus qui les contiennent. Toutefois, ne pas tenir compte des processus ouvre la porte à de potentiels vols de cycles par les usagers (voir l'encadré). Il est aussi envisageable de ne planifier que des processus sans considérer les fils. Cela revient à des fils de niveau usager car la planification des fils est sous la responsabilité du processus.

Vol de cycle par les fils d'exécution

Sur un système ne planifiant que les fils sans se préoccuper des processus, il est facile pour un usager d'abuser de la situation et de «voler des cycles» du processeur (ce qui correspond à augmenter sa priorité).
En effet, pour augmenter son temps UCT, il n'a qu'à créer un maximum de fils d'exécution à l'intérieur de son processus. Comme tous les fils sont égaux, ceux-ci auront tous une tranche de temps équivalente. Ainsi un utilisateur créant 10 fois plus de fils profitera de 10 fois plus de temps UCT.

Finalement, un mot sur le dernier cas, soit la planification à deux niveaux. Selon cette forme, les fils d'un même processus sont regroupés à l'intérieur d'une même entité (appelée processus, groupe, application ou autrement selon les environnements) administrée par le système d'exploitation. Ce dernier accorde alors une tranche de temps à chacune des «entités» (processus), tranche qui pourra ensuite être divisée entre les fils. De cette façon aucun processus ne sera en mesure d'augmenter artificiellement sa priorité (par vol de cycle).

Autre difficulté à analyser avec la planification sur multi-processeurs est celle de déterminer la façon d'effectuer la planification elle-même. Sur un mono-processeur, on planifie sur une dimension, i.e. on choisit le prochain processus dans la liste et on l'assigne à l'unique processeur. Sur un système multi-processeurs, la planification se fait sur deux dimensions : on doit choisir le prochain processus à exécuter et le processeur sur lequel l'exécuter. Cette seconde dimension (le choix du processeur) entraîne certains choix de conception tels que :

- **assignation statique ou dynamique ;**

  Une assignation statique signifie qu'un processus ou des fils d'exécution sont assignés à un processeur pour toute la durée de leur exécution. Ce type d'assignation entraîne peu de surcharge de travail et n'implique qu'une planification à court terme sur chaque processeur. Une planification de groupe s'intègre parfois à cette approche. Toutefois une mauvaise répartition de la charge de travail entre les processeurs s'avérerait difficile à corriger.

  Une assignation dynamique signifie que l'exécution d'un fil peut migrer d'un processeur à l'autre dynamiquement. Ainsi, lorsqu'un fil est en attente de reprise de son exécution, il attend le premier processeur disponible. Pour parvenir à ce partage, soit on a recourt à une file d'attente unique pour tous les processeurs, soit on fait passer dynamiquement le fil d'une liste à l'autre si on gère une liste d'attente par processeur.

- **utilisation de la multi-programmation ou non ;**

  Si nous nous fions à l'expérience sur mono-processeur, la multi-programmation [3] est primordiale pour assurer un usage optimal du processeur. Si elle n'est pas présente, un processus effectuant une E/S monopolisera l'UCT et cela même s'il ne l'utilise pas. Une meilleure gestion de l'UCT, grâce à la multi-programmation, garantissait dans ce cas un meilleur temps d'exécution moyen pour tous.

  Malheureusement cette affirmation n'est plus nécessairement vraie sur un multiprocesseurs. Même que dans ce cas particulier, il arrive que ce soit l'effet contraire qui soit obtenu, i.e. un ralentissement des exécutions. Il faut se souvenir que l'objectif visé par la planification n'est pas que tous les processeurs soient continuellement occupés mais plutôt de fournir une performance moyenne «quasi-optimale» pour l'exécution des applications. Ne pas avoir recourt à la multiprogrammation signifie que l'on assigne un processus à un processeur et qu'on le laisse exécuter sans intervenir jusqu'à ce qu'une réquisition soit nécessaire (fin de sa tranche de temps, arrivée d'un processus plus prioritaire, ...). Il monopolise donc l'UCT même lorsqu'il effectue une E/S.

- **Existence ou non de l'algorithme de planification court-terme.**

  Si on élimine la multiprogrammation, alors la question qui surgit naturellement est pourquoi ne pas aussi éliminer entièrement la planification à court terme, évitant ainsi une certaine surcharge due à cet algorithme ? En fait, même si la planification à court terme est toujours présente, il est fort probable que les algorithmes complexes déjà introduits (au cours de système d'exploitation) ne soient plus vraiment utiles. Une étude a même montré que le recours à des algorithmes complexes intégrant les concepts de priorité, de tranche de temps et de multiprogrammation n'offre pas un gain de performance significatif par rapport à l'algorithme FCFS dans un environnement multi-processeurs.

  La raison de ce changement est principalement due au fait que de laisser un processus monopoliser un processeur (sur un ensemble) s'avère souvent moins coûteux que de lui retirer. Il est généralement plus efficace d'attendre qu'un nouveau processeur se libère (s'il n'y en a pas déjà un de libre) que d'effectuer un changement de contexte. En effet, pourquoi réquisitionner un processeur (à la fin d'une tranche de temps par exemple) quand d'autres sont disponibles pour répondre à la demande. Ce serait là une démarche inutile qui en réalité ralentirait l'exécution du processus (car il reprendrait son exécution immédiatement de toute façon).

Dans les prochaines sections, nous décrivons plusieurs algorithmes visant à tirer au maximum des nombreux processeurs présents dans l'environnement.

Algorithme 1 : Partage de charge (Load sharing ou Time sharing)

Selon cet algorithme, on utilise une seule file globale de processus prêts pour tous les processeurs. Un fil n'est pas assigné à un processeur en permanence. Un processeur inoccupé choisit tout simplement le premier fil en attente dans la liste. La figure 7.18 illustre le fonctionnement de cet algorithme dans un environnement de 8 processeurs (ou cœurs). La file des processus prêts comprend cinq niveaux de priorités, un étant la priorité la plus élevé. Supposons que le processeur #4 devienne disponible, il choisit le processus «A» et l'exécute. Lorsque le processeur #3 deviendra disponible, il choisira d'exécuter le processus «B».

3. La multi-programmation consiste à donner le contrôle à un autre processus quand un processus se met en

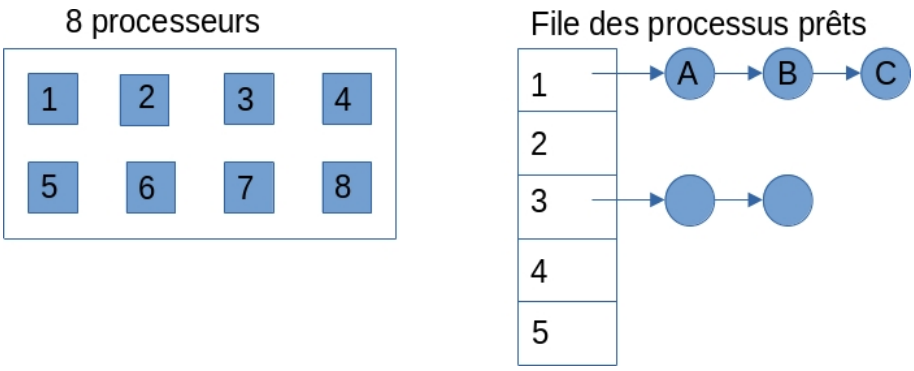

Figure 7.18 – Algorithme de partage de charge

Les avantages de cet algorithme sont multiples :

- la charge est balancée automatiquement ;
- la planification est décentralisée (pas de planificateur unique) ;
- la liste se trie selon la politique désirée (priorité, temps d'exécution, temps d'attente, ...).

Les principaux inconvénients de cette approche sont des difficultés probables pour accéder à la liste lorsqu'il y a un grand nombre d'UCTs. La liste devient alors un goulot d'étranglement. Une autre difficulté de ce mode de fonctionnement est celle d'entraîner une surcharge potentielle de travail due aux multiples changements de contextes (fin de tranche de temps, E/S, réquisition).

Des comparaisons ont été faites entre divers algorithmes gérant une liste unique de processus prêts :

- FCFS
  Selon cet algorithme, lorsque les processus (ou les fils d'exécution) démarrent (ou reviennent s'exécuter après une E/S) ils sont placés dans la liste dans l'ordre d'arrivée. Lorsqu'un processus (ou fil) est choisi, il s'exécute sans interruption jusqu'à ce qu'il lance une E/S ou termine.
- Plus petit nombre de fils en premier sans réquisition ;
  Dans ce cas, une priorité plus grande est donnée au processus qui possède le plus petit nombre de fils d'exécution. Encore une fois, lorsqu'un fil prend le contrôle d'un processeur, il s'exécute sans interruption jusqu'à sa prochaine opération d'E/S ou jusqu'à ce qu'il termine.
- Plus petit nombre de fils avec réquisition.
  Cet algorithme, comme le précédant, accorde une plus grande priorité aux processus possédant le plus petit nombre de fils d'exécution. Toutefois, lorsqu'un processus plus prioritaire revient dans la file d'attente, le planificateur réquisitionne un processeur, si aucun n'est disponible. Sinon un fil s'exécute sans interruption jusqu'à ce qu'il initie une E/S ou termine.

La simulation de ces trois algorithmes a montré que FCFS obtient les meilleurs résultats. Il semble donc inutile de faire appel à des algorithmes complexes.

Il est possible d'améliorer la performance des algorithmes en tenant compte du fait que sur un système multi-processeurs, les changements de contexte ont des propriétés indésirables, propriétés que l'on ne retrouve pas sur les systèmes mono-processeur :

attente sur une entrée/sortie.

1. Verrou d'attente active («spinlock») ;

   Sur un système multi-processeurs, existe l'éventualité qu'un fil perdant le contrôle de l'UCT (changement de contexte) détienne un verrou d'attente active. Les autres UCT en attente de ce verrou «boucleront» donc en vain. Cela ne se produit pas sur un mono-processeur car il n'y a pas de verrou d'attente active.

   La solution à ce problème consiste à faire une planification intelligente («smart scheduling»). Selon cette approche, un fil qui détient un verrou d'attente active ne se fait jamais retirer l'UCT tant qu'il détient le verrou (on lui alloue plus de temps).

2. Affinité ;

   Lors de la planification de nouveaux fils, les processeurs ne sont pas tous égaux. Si un fil «A» s'exécute sur le processeur P1, la cache de P1 contient toutes les informations pour exécuter A. Si «A» doit poursuivre son exécution très prochainement, il serait probablement beaucoup plus efficace de choisir de nouveau P1 pour l'exécuter. Avec un peu de chance, les informations de «A» seront toujours dans la cache et le système n'aura pas à les recharger à partir de la mémoire centrale. Le même raisonnement s'applique également pour la TLB qui sert à gérer la mémoire virtuelle.

   Les algorithmes qui tiennent compte de ces critères implantent la planification par affinité. L'idée de base est de toujours tenter de poursuivre l'exécution d'un fil sur le même processeur, soit celui de sa dernière assignation (sa dernière tranche d'exécution).

   L'approche par affinité a donné naissance à des algorithmes à deux niveaux de planifications. Pour ceux-ci, on utilise deux listes : une liste globale et une liste locale pour chacun des processeurs.

   - Lors de sa création, un processus est placé dans la liste globale. Il est ensuite assigné au processeur le moins chargé.
   - Lorsque le processus perd le contrôle, il est assigné à la liste locale du processeur sur lequel il s'exécutait ;
   - Ensuite, le processus sera assigné en priorité au même processeur grâce à un algorithme local œuvrant sur la liste locale ;
   - Si un processeur ne possède aucun fil dans sa liste, il ira chercher des fils dans les listes des autres processeurs.

   Il y a trois avantages à la planification à deux niveaux :

   (a) la distribution de la charge est équitable ;

   (b) la planification respecte l'affinité des processus ;

   (c) la liste globale risque moins de devenir un goulot d'étranglement car chaque processeur accède principalement à sa liste et l'accès aux autres listes ou à la liste globale est moins fréquent.

   Cette approche est principalement utilisée dans le noyau Mach avec les concepts de «local queue» et «global queue».

Algorithme 2 : Partage de l'espace

Quand un processus possède plusieurs fils d'exécution qui doivent interagir fréquemment (un groupe), il est avantageux de les exécuter simultanément pour faciliter et optimiser la communication. C'est ce que l'algorithme de partage de l'espace propose.

Cet algorithme fonctionne de la façon suivante :

1. Quand un groupe de fils d'un processus est créé, le planificateur vérifie si le nombre de processeurs disponibles est en mesure de satisfaire la demande ($\geq$ nombre de fils) ;
2. Si oui :
   - chaque fil est alloué à un processeur dédié et son exécution débute ;
   - chaque fil maintient son UCT jusqu'à ce qu'il termine. Cela signifie que si un fil bloque, il conserve l'usage du processeur et ce dernier demeure inactif jusqu'à ce que le fil débloque ;
   - Lorsqu'un fil termine, son processeur est retourné dans la liste de ceux qui sont disponibles.
3. Si non, aucun des fils n'est exécuté tant qu'il n'y a pas assez de processeurs disponibles.

La figure 7.19 illustre un partitionnement particulier entre trois processus. Le partitionnement P1 héberge un processus ayant six fils d'exécution, le partitionnement P2 en héberge un ayant 8 fils d'exécution et le partitionnement P3 en héberge un avec trois fils.

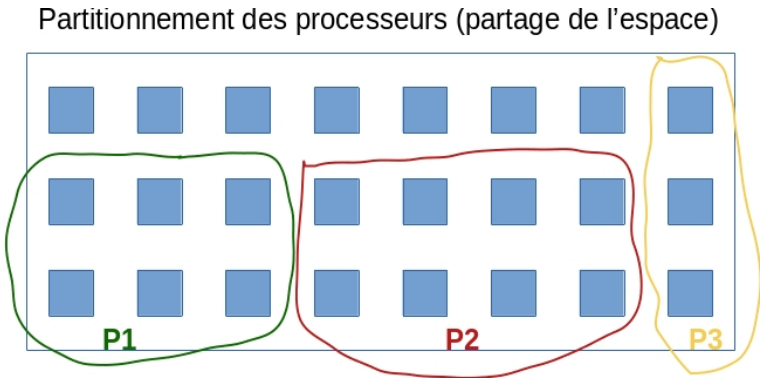

Figure 7.19 – Algorithme de partage de l'espace

Le planificateur alloue les processeurs aux groupes de fils d'exécution selon le premier arrivé premier servi (FCFS). Il est possible d'ajouter des priorités pour implanter des algorithmes tel le plus court d'abord (SJF). Cependant, l'expérience a montré que FCFS était presque toujours la meilleure solution.

Il est important de noter que dans un environnement comprenant plusieurs dizaines de processeurs, l'utilisation maximale de l'un d'entre eux n'est pas l'objectif le plus important. On vise surtout à tous les utiliser.

L'un des avantages de cet algorithme est l'absence de changement de contextes donc moins de surcharge de travail pour le système.

Cette première version de l'algorithme est un peu simpliste. En effet, s'il n'y a pas assez de processeurs disponibles, certains processus ne pourront pas s'exécuter. Pour corriger ce problème, on va permettre donc aux processus de varier dynamiquement leur nombre de fils, donc leurs besoins

en nombre de processeurs. Un algorithme, autorisant les processus à gérer leur degré de parallélisme, fonctionnera de la façon suivante :

- Chaque processus fixe le nombre minimal et maximal de processeurs dont il a besoin (soit le nombre de fils qu'il veut exécuter en parallèle) ;
- Le planificateur vérifie s'il y a suffisamment de processeurs pour satisfaire le processus (entre min et max) ;
- si oui :
  - chaque fil est alloué à un processeur dédié et son exécution débute ;
  - chaque fil maintient son UCT jusqu'à ce qu'il termine. Cela signifie que si un fil bloque, il conserve l'usage du processeur et ce dernier demeure inactif jusqu'à ce que le fil débloque ;
  - Lorsqu'un fil termine, son processeur est retourné dans la liste de ceux qui sont disponibles.
- si non, aucun des fils n'est exécuté tant qu'il n'y a pas suffisamment de processeurs disponibles.
- Périodiquement, chaque processus ajuste son minimum et son maximum selon le nombre de processeurs disponibles ;

Par exemple, soit un serveur Web :

- Le serveur détermine qu'il a besoin entre 5 et 15 fils pour s'exécuter ;
- Le serveur dispose actuellement de 10 fils ;
- Une surcharge ponctuelle le force à descendre à 5 fils. L'ajustement consiste à libérer un processeur lorsqu'un fil termine plutôt que de lui confier de nouvelles tâches.

Finalement, en observant bien ce mode de fonctionnement, on constate que le problème d'allocation de processeurs ressemble à celui de l'allocation de la mémoire dans la gestion de la mémoire virtuelle plutôt qu'à l'allocation de l'UCT sur un monoprocesseur. La question qui se pose est donc à peu près la même que sur l'allocation mémoire : Combien de processeurs doit-on minimalement allouer à un processus pour que son exécution se passe correctement ? Ce problème est aussi analogue à celui du nombre minimal de pages à allouer à un processus pour que son exécution progresse (pas d'écroulement).

La solution est aussi similaire. On utilise un ensemble de travail d'activités (*activity working set*). Cet ensemble défini le nombre minimal d'activités qui doivent être planifiées simultanément afin que le processus progresse adéquatement. Si ce minimum n'est pas atteint, il y aura risque d'«écroulement de processeurs». Cet événement survient quand l'allocation des processeurs à des fils, dont les services sont requis, implique le retrait de ces mêmes processeurs à des fils toujours actifs (ou qui le seront de nouveau très bientôt).

Un autre problème éventuel, qui ressemble également à celui que l'on retrouve dans la gestion de la mémoire, est la fragmentation de processeurs. Cela survient quand des processeurs sont alloués ou en fonction, mais que le nombre de processeurs disponibles est insuffisant ou, que ceux-ci sont incorrectement organisés pour répondre aux demandes des processus en attente.

Algorithme : Planification de groupe

L'inconvénient de l'algorithme précédent est la perte potentielle de temps si des processeurs demeurent inactifs (pas de multi-tâches). L'algorithme de planification de groupe tente donc de remédier à cette éventualité en considérant l'espace et le temps dans sa gestion.

Une planification dans l'espace consiste à exécuter en même temps les fils qui communiquent intensivement. Cette simultanéité permet aux échanges de messages entre ces fils de se compléter

plus rapidement. En effet, ne pas faire une planification dans l'espace risque d'allonger les délais de communication considérant que du temps devra aussi être dévolu aux changements de contexte. La figure 7.20 met en évidence l'importance de planifier en même temps des fils qui interagissent. L'on y voit l'exécution d'un processus A ayant deux fils d'exécution, A_1 et A_2, communicant entre eux. À la figure 7.20 (a), la communication entre les fils du processus A est retardée étant donné qu'ils ne sont pas planifiés simultanément (le message est arrivé et reçu par le système, mais livré seulement lorsque le fil de A reprend le contrôle). La figure 7.20 (b) illustre le gain potentiel lorsque l'on effectue une planification de groupe.

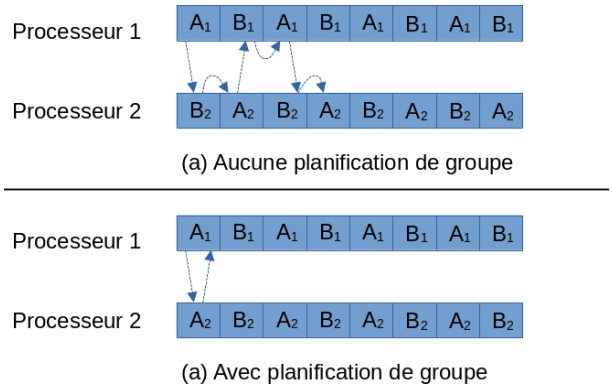

Figure 7.20 – Algorithme de planification de groupe

L'algorithme de planification de groupe se révèle en fait similaire à l'algorithme de co-planification (co-scheduling [75, 19, 127]) proposé en 1982 par Osterhout [75] avant même que ne soit introduit le concept de fils d'exécution. Il était alors appliqué à des processus communiquant sur le processeur CM*.

Pour bien fonctionner, la planification de groupe doit respecter les éléments suivants :

- Les fils interreliés (qui communiquent fréquemment) forment des groupes planifiés «ensemble» ;
- Tous les fils d'un groupe s'exécutent simultanément sur plusieurs processeurs en temps partagé ;
- Tous les fils d'un groupe débutent et terminent leur tranche de temps en même temps.

Ce qui est important à retenir, c'est que tous les processeurs doivent être planifiés de façon synchrone. L'idée est d'exécuter en même temps tous les fils d'un groupe. Ainsi :

- Le temps est divisé en tranches ;
- Au début d'une nouvelle tranche, on re-planifie de nouveaux fils sur chacun des processeurs ;
- Entre deux tranches, aucune planification n'est effectuée. Si un fil bloque, le processeur reste inactif jusqu'à la fin de la tranche du fil.

Les avantages de cet algorithme sont les suivants :

- minimiser les changements de contexte ;
  Soit deux fils d'exécution f_1 et f_2 devant se synchroniser. Si f_2 ne s'exécute pas (à l'état prêt) au moment où f_1 a besoin de se synchroniser, ce dernier devra se bloquer jusqu'à ce qu'un changement de contexte ramène f_2 à l'état actif.

- sauver du temps en allocation de ressources.
  Il permet notamment d'optimiser les accès simultanés aux fichiers.

L'algorithme de planification de groupe consiste en fait en un ensemble d'algorithmes. On peut le considérer tel un algorithme à plusieurs paramètres de configuration. Voici quelques exemples de son fonctionnement selon différents paramètres. Soit N processeurs et M processus chacun ayant un nombre k de fils tel que $k \leq N$. Les différentes options de planifications sont :

1. Chaque processus reçoit $1/N$ du «temps» sur les N processeurs et les tranches de temps sont les mêmes pour tous. La figure 7.21 illustre une telle allocation lorsqu'il y a quatre processeurs et quatre processus (P_1, P_2, P_3, P_4), chacun ayant respectivement 4, 1, 2 et 1 fils d'exécution.

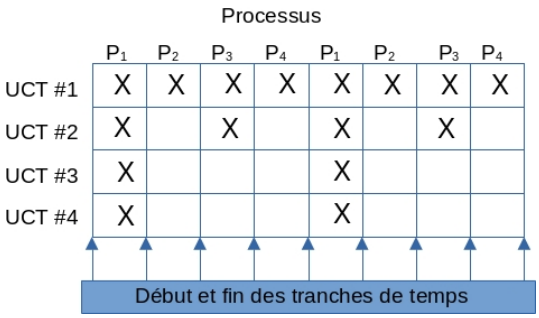

Figure 7.21 – Premier exemple d'algorithme de planification de groupe

   Le problème majeur de cet algorithme est son inefficacité. En effet, gérer plusieurs processus contenant peu de fils (voire un seul), est une situation qui provoque une lourde perte en temps de processeurs (UCT). On remarque d'ailleurs, dans l'exemple de la figure 7.21, qu'à chaque fois que P_2 et P_4 prennent le contrôle, trois processeurs demeurent inactifs.
2. Chaque processus reçoit une tranche proportionnelle à son nombre de fils d'exécution. La figure 7.22 présente une telle planification. Notons que le processus P_1 reçoit une tranche de temps quatre fois plus longue que celle de P_2. Toutefois, la perte en temps UCT, même si moindre que dans le cas précédent, est toujours présente.

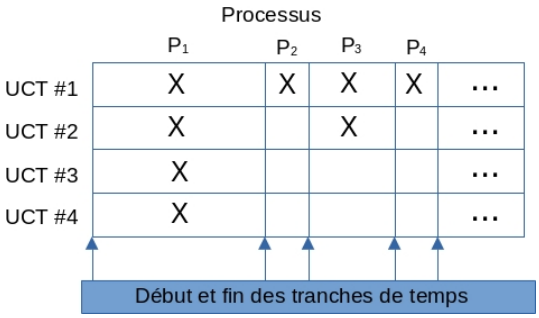

Figure 7.22 – Deuxième exemple d'algorithme de planification de groupe

3. Selon le nombre de processus et de processeurs, on peut tenter d'exécuter plusieurs processus à l'intérieur de la même tranche de temps, tout en respectant le concept de planifier tous les fils

d'un même processus ensemble pour éviter les délais de communication ou de synchronisation. La figure 7.23 présente un exemple d'une telle planification. Cette fois, peu de temps processeur est perdu.

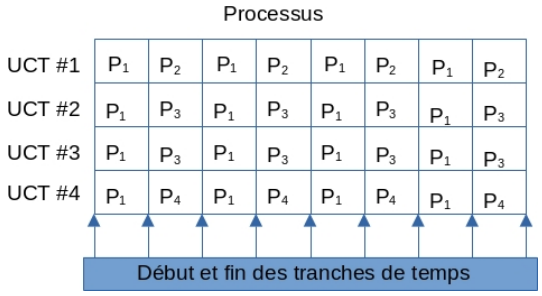

Figure 7.23 – Troisième exemple d'algorithme de planification de groupe

Lorsque le nombre de fils est supérieur au nombre de processeurs, il est nécessaire de planifier certains fils d'un même processus dans une autre tranche de temps. La figure 7.24 présente un exemple avec six processeurs et un processus, P_5, qui possède huit fils d'exécution.

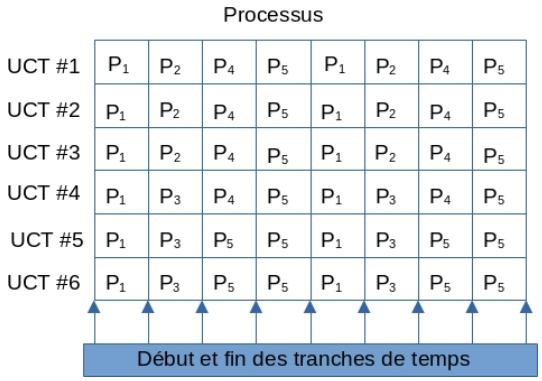

Figure 7.24 – Quatrième exemple d'algorithme de planification de groupe

Algorithme : Planification dynamique

Dans cet algorithme, les processus font des demandes d'allocation de processeurs au système. Ce dernier y répond selon le contenu de la requête et la disponibilité des processeurs. Les processus planifient alors eux-mêmes leurs fils sur les processeurs qui leur sont alloués.

Le mode de fonctionnement de l'algorithme est le suivant :

- Le système reçoit une demande d'allocation ;
  - Si un processeur est inactif et non déjà assigné → on l'alloue au demandeur ;
  - Si la demande provient d'un nouveau processus → on lui alloue au moins un processeur ;
  - Si aucun processeur n'est disponible → on conserve la demande.

- Le système reçoit une demande de libération.
  Le système ajoute les processeurs libérés à la liste de ceux disponibles. Il vérifie ensuite si des demandes sont en attente. Si oui, il alloue :
  - un processeur à chaque processus qui n'en a pas ;
  - les autres processeurs dans l'ordre FCFS.

Étude de cas

La plupart des systèmes d'exploitation modernes supportent les multi-processeurs et les multi-cœurs.

- Windows
  Le système d'exploitation Windows exécute les fils sur n'importe lequel des processeurs disponibles. Toutefois le répartiteur implante deux formes de planification par affinité :
  1. Affinité souple ;
     Le planificateur essaie toujours d'exécuter les fils sur le même processeur.
  2. Affinité rigide.
     Le planificateur assigne un processeur à un fil. Ce dernier s'exécutera exclusivement sur ce processeur.
- Linux
  À venir...
- MacOs
  À venir...
- Solaris
  À venir...

7.4.2 Multi-ordinateurs (réseau d'ordinateurs, systèmes répartis ou distribués)

La planification sur multi-ordinateurs ou planification répartie, s'effectue sur un réseau d'ordinateurs. Elle est rendue nécessaire par le besoin d'utiliser adéquatement les ressources d'un réseau (grappe de calcul par exemple) comprenant parfois des milliers d'ordinateurs. La planification répartie diffère de celle sur les multi-processeurs par le fait que dorénavant, il n'y a aucune mémoire commune. Faire appel à une structure de données partagées, telle la liste des processus prêts, n'est donc plus une option pour la répartition des processus.

Un élément très particulier relié exclusivement à la planification répartie (absent sur les multiprocesseurs), est le fait qu'elle soit implicite ou explicite. Une planification explicite signifie que l'utilisateur décide lui-même du site d'exécution d'un processus et même si ce dernier migrera dynamiquement ou non. Cette approche est généralement basée sur un modèle client/serveur et exige de l'environnement, une interface de programmation (API) explicite et précis. Les agents mobiles [76, 84, 136] sont un exemple d'environnement fournissant ce type de planification. On y retrouve parfois des commandes telle «`migrate(site_visé, ...)`» qui permet à un agent de migrer vers le site visé.

Comme la planification explicite n'exige en fait aucun algorithme de planification, nous n'abordons dans cette section que la planification implicite, i.e. une planification transparente à l'usager. Les objectifs de ce planificateur étant toujours les mêmes (temps réponse, bonne répartition de la charge), il faut y ajouter les considérations suivantes pour leur conception :

- Partage d'information
  Le partage d'information sur un réseau est plus complexe, non instantané et surtout plus coûteux. Il est généralement avantageux d'exécuter un processus localement si le temps pour obtenir les informations devient trop long.
- Coût de migration
  Avant de décider d'exécuter un processus sur un autre site, une analyse des coûts de ce transfert s'avère judicieuse.
- Coût de communication
  Exécuter des processus sur différents ordinateurs implique généralement des coûts de communication non négligeables si ceux ci doivent collaborer. Idéalement, les processus qui interagissent de façon significative ou partagent de l'information devraient s'exécuter sur un même site. Cette situation met en évidence le fait pour un planificateur de privilégier le recours à des processus plutôt qu'à des fils d'exécution.

Certains algorithmes conçus pour les multiprocesseurs peuvent toutefois s'adapter. C'est le cas de l'algorithme de planification de groupes. En adoptant une version sans multi-tâches de cet algorithme, il s'emploie comme planificateur long terme sur une grappe de calcul.

Évidemment, pour tenir compte des nouvelles contraintes, d'autres algorithmes ont été développés spécifiquement pour les multi-ordinateurs. Ces algorithmes seront tenus de :

- déterminer le site d'exécution du processus ;
  Les objectifs d'un algorithme de planification réparti sont principalement de balancer la charge de travail et d'obtenir un temps de virement minimal. Étant donné la difficulté et le coût pour migrer un processus, le choix du processeur pour l'exécution du processus est extrêmement important (beaucoup plus que pour les multiprocesseurs). L'algorithme doit donc choisir soigneusement les sites d'exécution du processus.
- fournir une approche statique ou dynamique ;
  Si la planificateur fournit une **approche statique**, le site d'exécution est déterminé à la création et le processus s'exécute en permanence sur ce site. Dans ce cas, il est primordial que la planification choisisse adéquatement le site puisqu'il n'y a plus de retour en arrière. Un bon choix permet de rencontrer les objectifs d'un tel algorithme soient de :
  - bien balancer la charge de travail afin de minimiser les pertes de cycle UCT (des processeurs inoccupés) et d'assurer l'équité (que la charge soit bien répartie entre les processeurs évitant ainsi les surcharges locales) ;
  - minimiser les communications.

  Pour faire leur choix, les planificateurs se basent sur les informations suivantes :
  - le besoin en temps processeur ;
  - le besoin en mémoire ;
  - le besoin en communication avec les autres processus.

  Plusieurs environnements modernes offrent la **planification statique** de processus. Des exemples de ceux-ci (planificateurs long terme), développés pour les grappes de calcul, sont Maui [134], PBS/OpenPBS [143, 71],Moab [18], Torque [147] et Slurm [118, 145, 88].
  Si la planificateur fournit une **approche dynamique**, il est possible que le processus change de processeur (site) pendant son exécution. Le choix du site de création est alors moins critique vu la possibilité de migrer un processus en cas de surcharge locale. Cependant l'algorithme qui se charge du choix du nouvel emplacement ainsi que de la migration du processus lui-même est complexe. Encore une fois, plusieurs facteurs sont à considérer pour migrer un processus :

  - le temps de transfert du processus
    Si le temps d'exécution restant du processus n'est que légèrement supérieur au temps de transfert, ce dernier est possiblement superflu.
  - La surcharge
    La surcharge de travail nécessaire pour migrer des processus est aussi à considérer. De même, il faut prévoir la possibilité que plusieurs migrations vers le même site se fassent simultanément, provoquant dès lors une surcharge locale sur ce site.
  - communication inter-processus
    On doit analyser les communications d'un processus avant de le migrer. S'il communique intensivement avec d'autres processus locaux, alors la migration sera probablement coûteuse en communication (bande passante sur le réseau) en plus de ralentir l'exécution du processus par des délais de communication.
  - utilisation de ressources
    En migrant un processus qui utilise des ressources locales, l'accès à celles-ci deviendra à la fois plus complexe et surtout plus lent. Cela provoquera encore une fois un ralentissement pour l'exécution du processus.
  - Autres...
    D'autres éléments déterminés par des particularités de l'environnement affectent aussi occasionnellement le choix de migrer ou non un processus.

  Malgré les coûts de migration d'un processus, certains systèmes offrent cette capacité. C'est le cas des systèmes Mosix [113, 137, 64, 1], OpenMosix [141], LinuxPMI [59]. Kerrighed [119, 51] et V [148, 17].
- identifier de façon unique un processus ;
  Le fait qu'un processus s'exécute sur un autre site (ou même plusieurs autres si la migration dynamique est possible), implique de prendre des décisions concernant la gestion de son descripteur (PCB) et de fournir un moyen de l'identifier de manière unique sur le réseau entier.
  Lorsqu'un processus migre, que faire de son descripteur ? Il existe plusieurs possibilités :
  - une seule copie sur le site de départ ;
    Cette approche est peu viable. En effet, le contexte se trouve sur une machine et le descripteur sur une autre. Un gestionnaire doit donc communiquer en permanence avec son «homologue» sur le site d'origine, d'où la nécessité constante d'avoir un gestionnaire sur chaque site. Ces gestionnaires doivent utiliser un protocole de type pair-à-pair pour parvenir à manipuler les descripteurs à distance.
    Cette approche est en fait peu efficace. Pour le devenir, un descripteur minimal serait requis sur le site d'exécution.
  - une seule copie sur le site d'exécution.
    Cette approche fonctionne à condition que le descripteur contienne de l'information sur le site de départ. Cette information est particulièrement utile pour, par exemple, retourner les résultats.
  - un copie complète sur chaque site ;
  - une copie partielle sur chaque site.

  De plus, sur la plupart des système d'exploitation les processus possèdent généralement un identificateur qui n'est unique que localement sur un site. Sur un système réparti, les processus pouvant migrer, on doit choisir une façon d'identifier uniquement les processus mais ce, à travers tout le réseau. Les différentes techniques pour générer des noms uniques sont ;

– de combiner l'identificateur du site et l'identificateur de processus (Pid) local ;
– d'employer un nom global choisi parmi une liste associée à chaque site.

Approche statique 1 : Algorithme basé sur les graphes

Cet algorithme offre une planification statique. Il est basé sur la théorie des graphes et suppose que l'utilisation de la mémoire et de l'UCT par le processus est connue à l'avance. Il suppose aussi l'existence d'une matrice spécifiant le niveau de communication entre chaque processus.

Le principe de l'algorithme consiste à assigner un sous-ensemble de processus à chaque processeur, évidemment si le nombre de processus est plus grand que le nombre de processeurs. Cette affectation doit minimiser le trafic sur le réseau.

Pour y parvenir, le système est représenté par un graphe pondéré G_p comprenant P nœuds dans lequel chaque nœud correspond à un processus et chaque arc au flux des messages entre deux processus. On fait l'hypothèse que le réseau sous-jacent est aussi décrit par un graphe G_o dans lequel les nœuds (au nombre de K) sont les ordinateurs et les arcs, les liens de communication. Mathématiquement, le problème se réduit à trouver une façon de partitionner le graphe G_p en K sous-graphes disjoints selon certaines contraintes (temps UCT et utilisation mémoire $<$ limite sur les ordinateurs du réseau) afin que l'on puisse le projeter sur le graphe G_o, le réseau d'ordinateurs. Pour chaque solution qui respecte les contraintes,

- les arcs internes à un sous-graphe réfèrent aux communications intra-machine et peuvent être ignorés ;
- les arcs se dirigeant d'un sous-graphe vers un autre forment le trafic sur le réseau.

Le but de l'algorithme est d'identifier un partitionnement qui minimise le trafic tout en respectant les contraintes.

Illustrons le fonctionnement de l'algorithme à l'aide d'un exemple. La figure 7.25 présente un graphe comprenant neuf processus (A, B, ..., I) et illustre le flux de communication entre chacun des processus. Ce graphe est projeté sur un réseau comprenant trois ordinateurs. La figure 7.26 propose un exemple de partitionnement. On remarque dans ce partitionnement un trafic total dans le réseau de 30 «unités». La figure 7.27 soumet un second partitionnement possible. Dans ce second exemple, le processus «C» est déplacé du nœud 3 vers le nœud 2. Selon ce nouveau partitionnement, le trafic total sur le réseau est de 28 «unités». Notons toutefois que le nœud 2 pourrait subir une surcharge de travail. La figure 7.28 suggère un troisième partitionnement possible dans lequel le processus «F», par rapport au premier partitionnement, est déplacé du nœud 2 vers nœud 3. Dans ce cas, le trafic total sur le réseau est de 19 «unités». Encore ici, il est possible que le nœud 3 soit surchargé.

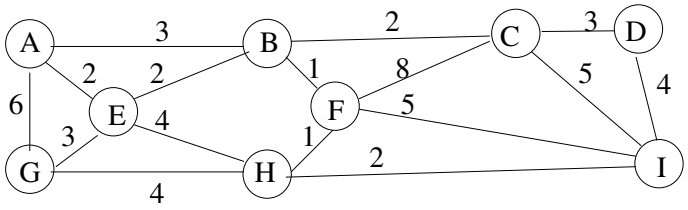

Figure 7.25 – Exemple de graphe de processus

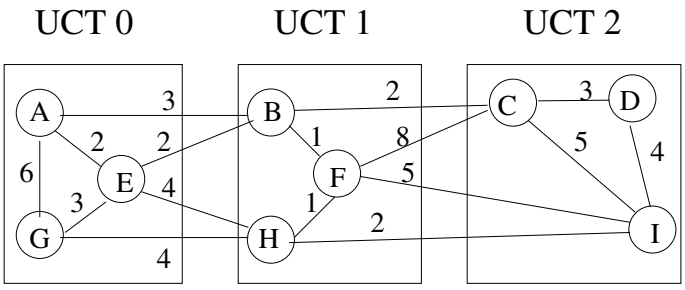

Figure 7.26 – Premier partitionnement

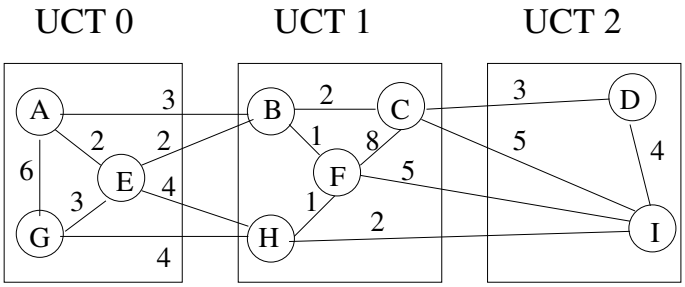

Figure 7.27 – Second partitionnement

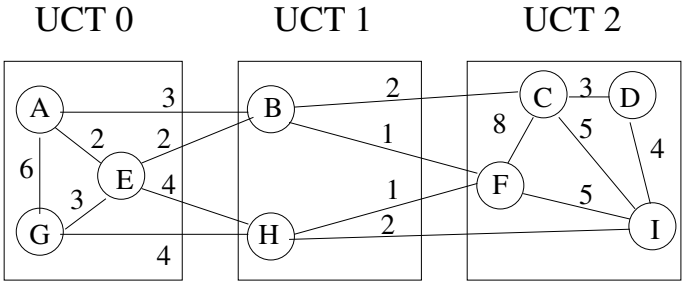

Figure 7.28 – Troisième partitionnement

Approche statique 2 : Algorithme basé sur les heuristiques et initié par l'expéditeur

Cette algorithme fait appel à des heuristiques simples pour déterminer si un processus doit s'exécuter localement ou non. La décision à savoir si un processus s'exécute localement ou non est prise par le système local (l'expéditeur potentiel) à chaque création de processus. Ainsi, lors de la création d'un processus,

- le site vérifie la charge locale .
- si la charge dépasse une certaine limite, le processus est exécuté localement (le site n'est pas surchargé) ;
- si la charge est au dessus de cette même limite (le site est surchargé), alors :
  - un autre site «A» est choisi au hasard et sa charge est demandée ;
  - si la charge de «A» est inférieure à une certaine limite, le processus est créé sur «A» ;
  - si la charge de «A» est supérieure à la limite, un second site est choisi au hasard et sa charge est demandée ;
  - Si aucun site n'est choisi après N tentatives, le processus est exécuté localement

Cet algorithme fonctionne très bien en général. Cependant, sous une lourde de charge, il génère un trafic intense. En effet, comme la charge est élevée sur tous les sites, ceux-ci ne cesseront de rechercher des candidats, mais tous ces échanges resterons vain.

Approche statique 3 : Algorithme basé sur les heuristiques et initié par le récepteur

Selon cet algorithme, un site sous-utilisé requiert du travail. Ainsi, à chaque fois qu'un processus termine sur un site, ce dernier vérifie sa charge de travail locale :

- Si sa charge trop faible, il choisit un site au hasard et lui demande du travail ;
- S'il n'a trouvé aucun travail après N essais, il cesse temporairement de demander ;
- Il recommence lorsqu'un autre processus termine ou après un temps d'attente (si sa charge est toujours faible).

Comme le précédent, cet algorithme fonctionne relativement bien en général. Contrairement à l'algorithme précédent, il produit peu de messages lorsque la charge est élevée. En effet, comme la majorité des sites sont surchargés, ceux-ci ne demanderont aucun travail (ne générant ainsi aucun message relié au partage de la charge). Toutefois, sous un charge légère, il en va tout autrement car tous les sites étant sous-utilisés, ils ne cesseront de demander du travail et engendreront alors un important trafic.

Approche statique 4 : combinaisons de 2 et 3

Afin de compenser les inconvénients des deux algorithmes précédents, il est possible de les combiner et d'appliquer certaines optimisations. Ainsi, au lieu de questionner la charge constamment, il est préférable de se souvenir de la charge de certains sites. De plus, plutôt que de demander la charge seulement sous une charge lourde ou légère, on peut opter pour :

- que chaque site recueille régulièrement afin d'avoir des informations à jour ;
- que chaque site diffuse sa charge périodiquement.

Approche statique 5 : Algorithme basé sur les appels d'offre

Le principe de cet algorithme est de simuler un mini système économique en se basant sur des appels d'offre. Dans ce système, les joueurs sont les processus qui doivent acheter des services

(en particulier du temps UCT) et chaque site soumet des prix pour ses services. Les prix sont des indicateurs de la valeur du service (vitesse de la machine, quantité de mémoire, charge, temps réponse, autres ressources, ...).

Ainsi, lors du démarrage d'un processus, le système :

- recherche les sites fournissant le service requis ;
- identifie les sites que le processus «peut se payer» ;
- calcule le meilleur candidat (le moins coûteux) ;
- génère une offre et la soumet à son premier choix ;
- attend la réponse.

Les différents sites reçoivent régulièrement des offres. Il en choisit une et informe le gagnant. Les sites ayant soumis des offres non sélectionnées en sont également informés. Les prix de ce site sont ensuite mis à jour pour les prochaines soumissions.

Cet algorithme cause plusieurs problèmes dont le principal est le moyen de gérer «l'argent virtuel» nécessaire au le bon fonctionnement du système. D'où vient cet argent et comment l'obtient-on ?

Approche dynamique

Peu d'algorithmes ont été proposés pour gérer la migration dynamique de processus. Les algorithmes à long terme sont rarement sophistiqués dans un tel environnement puisque les mauvaises décisions (migration vers un mauvais site) sont aisément récupérables (par une seconde migration), donc tolérables.

Déterminer le moment et la façon de migrer dynamiquement un processus s'avère être un problème plus complexe pour lequel il y a peu d'algorithmes.

7.4.3 Exemples de systèmes et leur gestion de l'UCT

- bidding, échange d'états
- Mach
- Amoeba
- PBS, Torque/Maui, Torque/Moab
- Sun Grid
- Condor, Boinc
- Mosix
- Slurm
- ...

7.5 Gestion de la mémoire

La gestion de la mémoire concerne la mémoire centrale ainsi que celle du disque lorsque les notions de mémoire virtuelle, de pagination et de «swapping» sont supportées.

Rappelons que la gestion de la mémoire virtuelle exige un support matériel. Sur la plupart des ordinateurs, on retrouve le MMU (*Memory Management Unit*) dont le rôle consiste à gérer l'adressage (table de segments, table de pages) et la TLB (*Translation lookaside Buffer*) qui elle, constitue une forme de mémoire cache permettant d'accélérer la traduction d'adresses. Certains environnements [96, 97] fournissent aussi des contextes (tables de contextes).

Rappelons aussi deux concepts importants en mémoire virtuelle : la pagination et la segmentation. Selon les systèmes, ces concepts ne sont pas toujours définis de la même manière. Tel qu'illustré à la figure 7.29, la segmentation paginée découpe l'espace d'adresses d'un programme en respectant des entités logiques (code, données, pile, heap). Elles sont ensuite placées dans des segments sur lesquels on applique la pagination. La figure 7.29, quant à elle, présente la pagination segmentée qui considère plutôt tout l'espace d'adresses d'un programme comme une suite de pages de même taille qui sont regroupées dans des segments, aussi de tailles égales pour en faciliter la gestion.

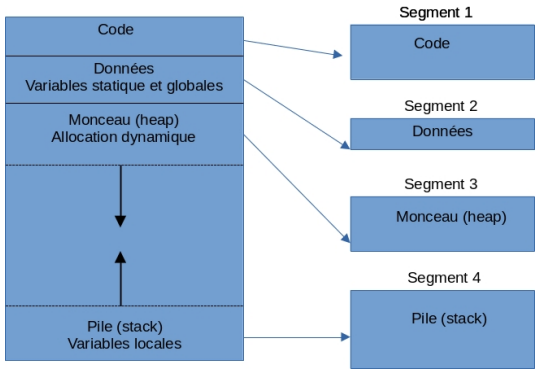

Figure 7.29 – Segmentation paginée

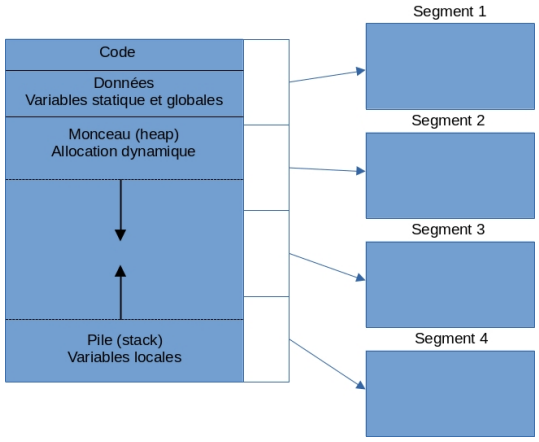

Figure 7.30 – Pagination segmentée

Une autre notion importante associée à cette gestion est celle de la mémoire partagée. Pour sauver de l'espace, le partage des segments de code (accessible en lecture seulement) est fréquemment implantés. Les bibliothèques partagées que l'on retrouve dans tous les systèmes sont des exemples de code partagé. On peut, par exemple, associer un segment à chaque bibliothèque et partager le segment. Des exemples de bibliothèques partagées sont les «.DLL» de Windows et les «.SO» du système Linux.

Il est aussi possible de partager des données modifiables. Dans ce cas, de multiples solutions existent et dépendent de l'environnement. Les solutions adaptées aux systèmes multiprocesseurs ne sont, en effet, pas les mêmes que celles sur un réseau d'ordinateurs. Malgré tout, dans les deux cas, il faut gérer les problèmes de cohérence (mais différemment).

7.5.1 Gestion de la mémoire sur multi-processeurs

Sur un système multiprocesseurs, il est assez simple de partager la mémoire. Toutefois, celle-ci s'avère rapidement être un potentiel un goulot d'étranglement comme nous l'avons abordé dans la section 7.1 dans le cas des architectures matérielles et ce, quel que soit le type d'architecture choisie (UMA ou NUMA). L'utilisation de la mémoire cache est généralisée afin d'accélérer les accès et d'éviter les goulots d'étranglement. Évidemment, la gestion de la cache et celle de la cohérence des données en cache se révèlent complexes. Nous y reviendrons.

7.5.2 Réseau d'ordinateurs

Sur un réseau, il n'y a aucune mémoire partagée. Tout échange d'information se fait par le passage de messages. Du fait que certaines personnes en charge du développement préfèrent le concept de mémoire partagée, il y eut plusieurs tentatives d'offrir l'illusion de mémoire commune sur un système distribué. On appelle ce concept «la mémoire partagée distribuée (*Distributed Shared Memory*) ou DSM. Plus formellement, la mémoire partagée distribuée [128] est une forme d'organisation logique de la mémoire dans laquelle des unités de mémoire physiquement distinctes (sur des ordinateurs séparés) sont adressables comme un seul espace d'adresses logiques. Dans ce cas particulier, le terme «partagé» ne signifie pas qu'il y a une seule mémoire centrale mais bien que l'espace d'adresses est partagé, i.e. qu'une même adresse sur plusieurs ordinateurs réfère à la même localisation en mémoire «logique». Dans certains contextes, on réfère aussi à cette façon de faire sous l'appellation d'espace d'adresses global distribué (Distributed Global Address Space ou DGAS).

Plusieurs solutions ont donc été proposées pour simuler de la mémoire partagée sur un réseau d'ordinateurs et la figure 7.31 résume les différentes architectures proposées dans la littérature. Ainsi, il est possible d'implanter la mémoire partagée au niveau matériel ou logiciel. Au niveau logiciel, la DSM est fournie soit par le système d'exploitation, soit par une bibliothèque ou soit par le langage de programmation.

Niveau matériel

Il y a eu plusieurs tentatives pour développer des unités matérielles offrant la DSM. Plusieurs d'entre elles consistaient à améliorer les capacités du réseau. L'une d'elle, Memnet [26], intégrait un réseau local développé pour fournir de la mémoire distribuée. En ce sens, plutôt que d'apparaître comme un périphérique d'E/S, Memnet apparaissait comme une zone mémoire dans l'espace d'adresses de chacun des ordinateurs du réseau. CAPNET [102] a étendu ce concept à des réseaux plus larges.

D'autres ont développé des environnements plus complexes qui ressemblaient finalement plutôt à des multiprocesseurs de type NUMA. Les systèmes DASH [56], DDM [67, 42, 111] et PLUS [13], Galactica Net [162] fournissaient un unique espace d'adresses à tous les processeurs.

Protic et al. [78] et Eskicioglu et Marsland [30] détaillent ces concepts et citent plusieurs autres exemples d'implantations.

Niveau OS

Plusieurs approches ont été proposées pour intégrer la mémoire distribuée au niveau du système d'exploitation.

L'une d'entre elles consiste à partager des «pages» par l'intermédiaire de la mémoire virtuelle. Rappelons que la mémoire virtuelle permet d'exécuter un programme partiellement chargé en mémoire centrale. Elle sépare le programme en pages (1K, 2K, 4K ou 8K) à charger au besoin. La figure 7.32 décrit le fonctionnement de la mémoire virtuelle. Un processus s'exécute normalement tant qu'il ne tente pas d'accéder à du code ou à des données non localisés en mémoire centrale. Lorsque cela se produit, une interruption de type «faute de page» est générée et initie le chargement du code ou des données manquantes. Lorsque le chargement prend fin, le programme poursuit son exécution. La figure 7.33 expose le fonctionnement de la mémoire virtuelle lorsque la pagination se fait à partir d'un disque distant (sur un serveur). Cette configuration était fréquente dans les années 1980 lorsque les stations de travail ne comportaient aucun disque local (diskless) dû à leur coût élevé.

S'il est possible de répondre à une faute de pages à partir d'un disque distant, il est également possible de la faire à partir de la mémoire centrale d'un autre ordinateur comme l'illustre la figure 7.34.

Le partage de pages à partir de la mémoire centrale a servi à implanter la migration dynamique de processus et la mémoire virtuelle distribuée ou DSVM (Distributed Shared Virtual Memory).

1. Migration de processus
   Le système d'exploitation expérimental «V» a recouru à cette fonctionnalité pour migrer des processus. L'unique action à accomplir par le système dans le but de migrer un processus consistait à transférer son descripteur et à re-démarrer son exécution. Au démarrage, il se produisait une faute de page, et alors le système de pagination à distance se chargeait de transférer les pages de l'ordinateur source vers le nouveau site.
2. Distributed Shared Virtual Memory (DSVM)
   Li et Hudak ont affiné ce système en créant la DSVM [103, 57]. Selon la DSVM, on partage une zone de la mémoire virtuelle (zone globale en mémoire centrale). Les pages de cette zone se situent toutes en mémoire centrale et sont distribuées sur différents sites. Ainsi chaque page est présente en mémoire centrale d'une des machines. Lorsqu'une machine fait référence à une page absente de sa mémoire, il se produit la séquence d'événements suivants :
   - faute de page ;
   - la page est localisée et une demande de transfert est envoyée ;
   - la page est transférée (marquée absente sur l'ancien site).

   La figure 7.35 illustre la distribution d'une mémoire virtuelle contenant 11 pages sur plusieurs sites. La figure 7.36 quant à elle, présente deux situations dans lesquelles une page, la page 6, est partagée entre deux sites. Dans le premier cas, la page est partagée en lecture seulement et est dupliquée sans risque sur les deux sites. Dans le second cas, la page est partagée en écriture et généralement transférée d'un site à l'autre pour effectuer les modifications.
   L'utilisation de la DSVM entraîne certaines problématiques, dont :
   - Qu'arrive-t-il si on tente de répliquer une page en écriture ?
     La situation est similaire à celle que l'on retrouve dans la gestion d'une cache. On doit impérativement s'assurer de la cohérence des données. Il existe plusieurs solutions :
     – Élimination des copies ;
       Ainsi, lorsqu'une page partagée est sur le point d'être modifiée, un signal demandant

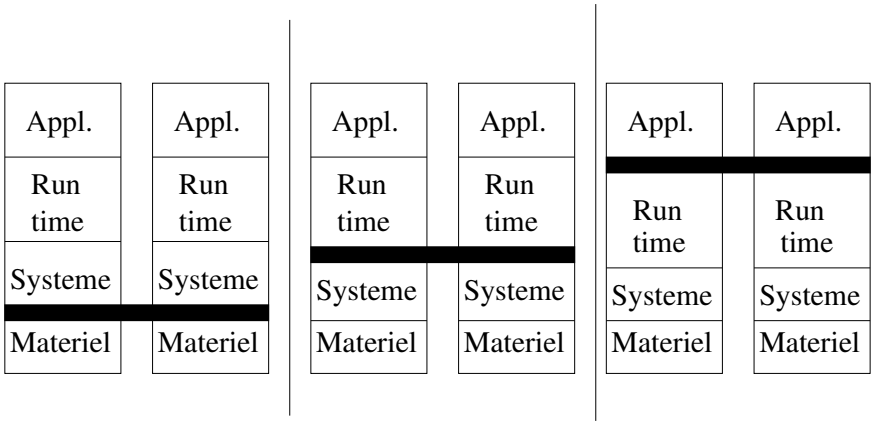

Figure 7.31 – Archtectures pour la mémoire partagée distribuée

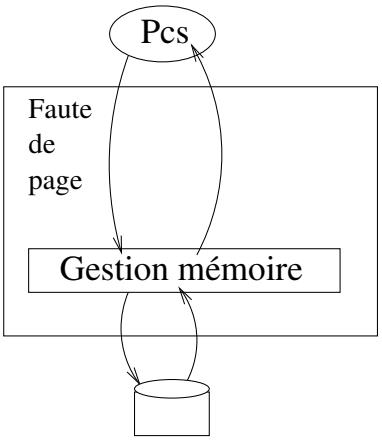

Figure 7.32 – Fonctionnement de la mémoire virtuelle

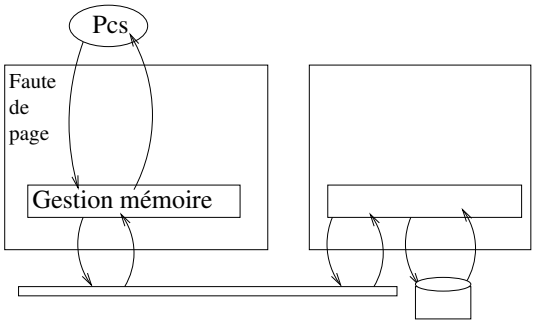

Figure 7.33 – Fonctionnement de la mémoire virtuelle sur un disque distant

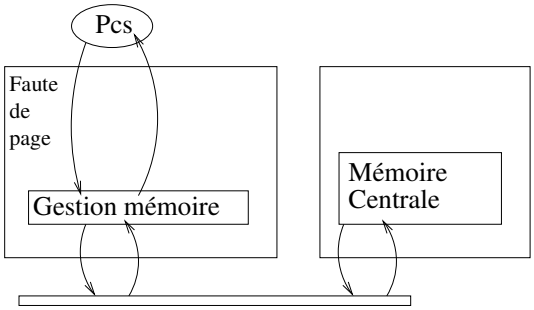

Figure 7.34 – Fonctionnement de la mémoire virtuelle à partir de la mémoire centrale d'un autre ordinateur

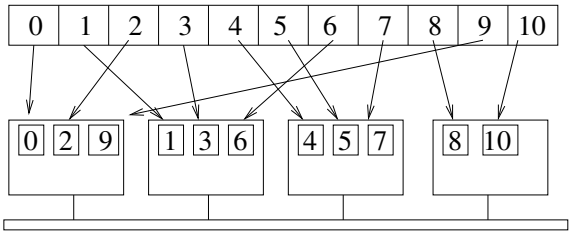

Figure 7.35 – DSVM : distribution des pages sur différents sites.

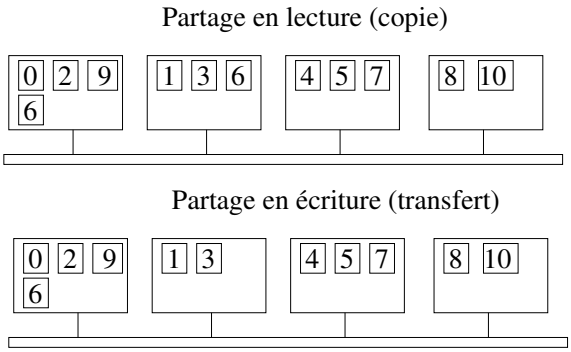

Figure 7.36 – DSVM : distribution des pages sur différents sites.

la destruction de la page est envoyé à tous les autres sites possédant une copie. Lorsque tous les sites ont répondu (ont détruit leur copie), la page originale est alors mise à jour.
Cette solution rend cependant les écritures très lentes et affectera également la performance des futures lectures sur les sites ayant détruit leur copie.

– Verrouillage ;
Lors de la mise à jour d'une donnée, un processus acquiert un verrou sur une portion de la mémoire virtuelle (la page) avant d'effectuer son traitement. À la libération du verrou, les changements sont propagés à toutes les copies. Si l'acquisition d'un verrou est réservée à une seule machine à la fois, alors la cohérence des données est assurée.
À noter que cette solution risque d'entraîner des problèmes de performance. Pour aider à les surmonter un certain nombre d'optimisations ont été proposées. Par exemple, lorsque des mises à jour sont apportées à une page, il convient de transmettre seulement les éléments altérés plutôt que la page complète. Pour y parvenir, une copie propre de la page est conservée sur la machine qui procède aux ajustements. C'est seulement alors que le verrou est acquis, les modifications effectuées puis que le verrou est relâché. Les mises à jour sont, soit transmises dès la libération du verrou, soit lorsqu'une autre machine tentera d'accéder à cette page. À ce moment, la machine ayant modifié la page, compare l'état courant de la page à la copie propre et ne transfère que les changements. La page distante est alors mise à jour.

Nous traiterons plus loin dans cette section de la façon de gérer la cohérence des données de manière plus efficace, en particulier en diminuant les exigences en terme de cohérence.

- Qu'arrive-t-il si on tente de transférer une page sur laquelle le site écrit activement ?
Dans ce cas, une baisse de performance des écritures est probable, en plus de forcer des transferts continus d'une page d'un site vers un autre. Un solution envisageable consiste à verrouiller une page et à interdire le transfert tant que les opérations ne sont pas terminées. Cette approche influencera toutefois la performance des autres sites désirant écrire sur cette page.
- Problème du faux partage.
Selon le concept de DSVM, lorsqu'une zone de mémoire partagée est référencée, un bloc mémoire entier (une page ou plus) contenant la zone visée est transféré. Quelle est la taille de ce bloc ? Toujours selon le principe de DSVM, la taille d'un bloc doit être un multiple de la taille d'une page.
Le transfert d'un bloc mémoire est coûteux. Les temps de latence et de transfert s'avèrent relativement longs et peuvent ralentir de façon significative les accès aux données. Pour minimiser ces temps, on opte pour des blocs dont la taille est la plus grande possible car ainsi les transferts sont moins nombreux, d'où une utilisation plus optimale de la bande passante. De plus, le concept de localité indique que les accès en mémoire se situent toujours dans des zones voisines. L'usage de gros blocs évitera aussi de multiples transferts de blocs voisins et optimisera le temps de communication. Donc, plus le bloc est de grande taille plus la communication est efficace.

Définition : Latence

La latence est le temps de traitement nécessaire à l'environnement pour préparer l'envoi d'un paquet sur le réseau. Elle est aussi désignée par l'expression «*startup time*».

Il faut cependant savoir que l'emploi de «gros blocs» apporte un inconvénient majeur, soit le faux partage. Le faux partage est une situation qui se produit quand deux processus accèdent simultanément à des données qui n'ont aucun lien entre elles, mais sont situées dans la même région logique de la mémoire, une page par exemple. Ainsi, supposons que deux variables, V_1 et V_2, n'ayant aucun lien entre elles, soient logées dans la même page comme suggéré à la figure 7.37. Supposons aussi qu'un processus P_1 sur un site A_1 accède activement V_1 et qu'un processus P_2 sur un site A_2 utilise activement V_2. Dans cette situation, la page contenant ces deux variables sera continuellement transférée entre les deux sites (ou verrouillée). Ce transfert est inutile car en réalité le deux processus ne partagent aucune variable [4].

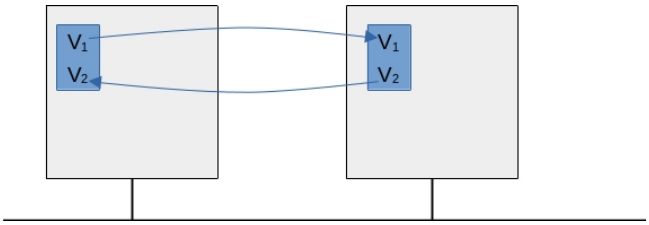

Figure 7.37 – DSVM : distribution des pages sur différents sites.

Pour conclure au sujet de la mémoire distribuée (niveau OS), celle-ci a été implantée dans quelques systèmes d'exploitation expérimentaux. Certains l'ont conçu comme une extension de la mémoire virtuelle, tandis que d'autres ont proposé des approches mixtes basées sur des modifications au système d'exploitation et une bibliothèque dans l'espace usager.

Niveau bibliothèque et langage

Certaines implantations [30, 78, 128] de la DSM ont été réalisées au niveau de bibliothèques ou de langages de programmation. Ces environnements présentaient l'avantage de faire appel à des concepts moins grossier que les pages, les concepts d'objet et de tuple en particulier, qui permettent de regrouper dans un espace logique seulement les données qui possèdent un lien entre elles (moins de faux partage).

7.5.3 Cohérence

Régulièrement, on mentionne que les programmes parallèles doivent communiquer entre eux fréquemment. La mémoire commune est un des principaux médiums de communication. Nous avons d'ailleurs présenté différents algorithmes parallèles utilisant des données partagées (en mémoire)

4. Cette situation se produit aussi dans d'autres contextes [28, 129] telle que la gestion de la mémoire cache.

comme moyen de communication, en particulier afin d'assurer l'exclusion mutuelle. Comme la plupart des algorithmes parallèles, nos exemples supposent que les données partagées en mémoire sont cohérentes. S'il n'existe qu'une unique copie de la donnée, il n'y aura aucun problème de cohérence. Toutefois pour maintes raisons (dont la performance), de multiples copies d'une même donnée sont présentes dans la plupart des environnements. Citons en particulier le cas de la totalité des ordinateurs modernes qui multiplient les copies des données par l'intermédiaire des différents niveaux de mémoire cache. Or, dans un système de gestion mémoire, qu'elle soit matérielle ou logicielle, centralisée ou distribuée, les modifications apportées aux différentes copies d'une donnée en mémoire (cache ou éloignée), ne sont pas toujours instantanées. C'est ainsi qu'éventuellement des incohérences se manifestent.

Définition : Cohérence mémoire [150]

La cohérence est la capacité pour un système à refléter sur la copie d'une donnée les modifications intervenues sur d'autres copies de cette donnée. Cette notion concerne particulièrement les systèmes de fichiers, les bases de données et les mémoires partagées.

La cohérence mémoire consiste donc à définir la façon dont les programmes parallèles observent l'état de la mémoire partagée.

Exemple 1 : incohérence sur les ordinateurs modernes

Sur tous les ordinateurs modernes, les processeurs contiennent plusieurs niveaux de cache. Les mises à jour des données par un processeur sont d'abord effectuées dans la mémoire cache. Cette nouvelle valeur demeure parfois un certain temps dans la cache avant d'être recopiée en mémoire centrale. Pendant cette courte période, la donnée est incohérente due au fait qu'elle est associée à au moins deux valeurs distinctes et incompatibles.
Ces incohérences, comme nous l'avons déjà montré, font que les algorithmes d'exclusion mutuelle par attente active (Peterson, Dijkstra ou Dekker) ne fontionnent plus sur ces ordinateurs.

Les incohérences introduites par le manque d'instantanéité d'une mise à jour crée une situation où les valeurs obtenues à partir d'une même adresse peuvent être différentes. Par exemple, un processus pourrait lire une valeur désuète comme l'illustre la séquence d'exécutions présentée la figure 7.38. Dans celle-ci, le temps «absolu» s'écoule de gauche à droite. On note que le processus P_2 retourne une valeur désuète, selon le temps absolu. En revanche, le processus P_3 retourne une valeur différente mais qui, elle, est à jour. Il y a donc incohérence entre les deux valeurs obtenues.

Temps $\longrightarrow$		
P_1	W(x)1	
P_2		R(x)0
P_3		R(x)1
Initialement x = 0		

Figure 7.38 – Exemple d'exécutions

Quelles sont les conséquences de telles incohérences (elles existent sur tous les ordinateurs modernes) ? On a déjà discuté de certaines d'entre elles aux sections 1.6 et 3.5, en particulier, celles provoquant le mauvais fonctionnement des algorithmes de synchronisation par attente active (Dijkstra, Peterson, ...). L'exemple 2 met en évidence une situation pour laquelle les sorties d'un programme parallèle (deux fils d'exécution en c++) sont imprévisibles.

Exemple 2 : incohérence dans les processus concurrents

Soit les deux programmes suivants qui s'exécutent en parallèle (tiré de [14]) :

```
    ...
(1) var1 = 1;
(2) cout << var2 << endl;
    ...
```

Programme 7.1 – Processus $P1$

```
    ...
(3) var2 = 1;
(4) cout << var1 << endl;
    ...
```

Programme 7.2 – Processus $P2$

Les sorties possibles de ces deux processus sont

- 01
  Le processus $P1$ s'exécute entièrement avant le processus $P2$ (ordre d'exécution des instructions : (1) → (2) → (3) → (4)), ou bien l'inverse, i.e. $P2$ s'exécute entièrement avant $P1$ (ordre d'exécution des instructions : (3) → (4) → (1) → (2))
- 11
  La première instruction de chaque processus s'exécute avant la seconde instruction (pour chaque processus). Les différents ordres d'exécution des instructions seraient :
  - (1) → (3) → (2) → (4),
  - (3) → (1) → (2) → (4),
  - (1) → (3) → (4) → (2),
  - (3) → (1) → (4) → (2).
- 00
  Ce résultat, qui ne semble pas respecter l'ordre d'exécution des énoncés d'un programme, est rendu possible par le fait que la mémoire est incohérente (soit que les nouvelles valeurs affectées à $var1$ et $var2$ sont conservées un certain temps dans la mémoire cache ou encore, que la propagation de leur mise à jour est retardée pour une raison inconnue).

Étant donné les problèmes que cela entraîne, pourquoi les responsables de la conception acceptent-ils cette situation ? En fait, la question est plutôt de savoir si «**cette situation est vraiment grave**», et si oui, à quel point ?

Les concepteurs/conceptrices de systèmes de gestion mémoire ont généré cette problématique tout simplement afin de parvenir à augmenter la performance des ordinateurs (ce qui est leur priorité no. 1). En fait, mettre tout à jour «simultanément» est, du point de vue performance, un réel désastre. Pour les responsables de la conception du matériel (et même ceux qui développent des compilateurs), ce phénomène est là pour rester et les personnes en charge de la programmation de ces machines devront faire avec !

En lien avec les programmes, la gravité des incohérences dépend de la façon dont ils sont conçus. Les algorithmes d'exlusion mutuelle par attente active (Dekker, Dijkstra et de Peterson) ne fonctionnent plus car ceux-ci supposent que la mémoire sous-jacente est toujours parfaitement cohérente,

en ce sens qu'elle respecte des contraintes très strictes d'ordonnancement des instructions en mémoire. Ces contraintes imposées sur l'ordre des opérations de lecture et d'écriture sur une mémoire partagée forment ce qu'on appelle un modèle de cohérence de la mémoire.

Définition : Modèle de cohérence

Un modèle de cohérence définit toutes les contraintes imposées sur l'ordre des opérations de lecture et d'écriture en mémoire partagée. Il indique donc l'ordre dans lequel les opérations en mémoire semblent être exécutées.
Le modèle de cohérence définit donc l'ensemble des valeurs qu'une lecture peut retourner.
Les modèles de cohérence sont importants car, sur les systèmes à mémoire partagée, plusieurs processeurs/cœurs accèdent simultanément à la même localisation en mémoire centrale. Bien comprendre les incohérences potentielles permet de mieux les gérer. Les personnes développant des programmes dans ces environnements se servent de ces modèles conceptuels pour assimiler la sémantique des opérations et manipuler correctement la mémoire partagée. Ceux-ci se doivent d'être intuitifs (si possible), faciles à utiliser (à programmer), performants et portables. Le hic avec des modèles trop simples et trop intuitifs est qu'ils affectent (négativement) la performance des architectures qui les implantent.

Le modèle de cohérence adéquat pour assurer le bon fonctionnement des algorithmes d'attente active est soit la mémoire atomique, soit la «cohérence séquentielle», deux modèles de cohérence dit forts. Cependant, comme déjà mentionné, la plupart des ordinateurs modernes ne respectent pas ces contraintes et fournissent des modèles de cohérence beaucoup plus faibles qui autorisent un ré-ordonnancement des opérations de lectures et d'écritures. Ainsi, toute opération de lecture ou d'écriture est sujette à un ré-ordonnacement par rapport aux autres lectures ou écritures aussi longtemps que cela n'affecte pas le comportement d'un seul fil d'exécution pris individuellement. On dit dans ce cas que le processeur (physique ou virtuel) possède un modèle de cohérence mémoire faible. Introduisons quelques notions de cohérence de la mémoire. Sans entrer dans les détails (pour l'instant), nous abordons les questions à savoir pourquoi et comment les environnements modernes (compilateurs et ordinateurs) ne respectent pas la cohérence séquentielle.

Il est important de préciser que la plupart des programmes ne seront jamais affectés par ces ré-ordonnancements (en particulier les programmeaucuns séquentiels). De fait, la règle respectée par toutes les formes de ré-ordonnancement est la suivante :

Thou shalt not modify the behavior of a single-threaded program [69].

De plus, même si la probabilité de son occurence n'est pas nulle, le fait de recourir à des outils de synchronisation, tels que les sémaphores, permet aux programmes de fonctionner en présence d'incohérences ou de les éviter. Il existe toutefois une catégorie d'algorithmes, faisant appel à des techniques de synchronisation sans verrouillage (algorithmes sans verrous), pour lesquels ces incohérences engendrent des situations problématiques. L'utilisation de ces techniques est toutefois réservé à du personnel hautement spécialisé.

Dans tous les cas, les personnes qui développent des programmes parallèles dans ce type d'environnement doivent bien comprendre les implications de leurs choix. Qu'est-ce qui est en jeu lorsqu'on accède à de la mémoire qui admet à l'occasion des valeurs désuètes et contradictoires ? Comment

Définition : Sémantique des opérations de lecture et d'écriture

Le modèle de cohérence permet de mieux assimiler la sémantique des opérations de lecture et d'écriture. Mais quelle est la sémantique de base de ces opérations ?
La sémantique de l'opération de lecture détermine principalement la valeur qu'elle doit retourner. Intuitivement, la lecture doit retourner la valeur de la dernière écriture (à cette même adresse). Sur un seul processeur, la dernière écriture est bien définie par l'ordre des opérations du programme. On suppose donc que chaque opération s'exécute une à la fois dans l'ordre spécifié par le programme. C'est un modèle simple et intuitif et qui a l'avantage de s'implanter facilement.
Le modèle de cohérence caractérise très préciséement le comportement de la mémoire en relation avec les opérations de lecture et d'écriture. Il spécifie donc les valeurs qu'une lecture peut retourner.
Différentes optimisations sont alors concevables avec un tel modèle. Ainsi, par exemple, c'est suffisant d'exécuter des opérations dans l'ordre imposé par le programme, mais seulement s'il y a des dépendances de données ou de contrôle (i.e. quand deux opérations accèdent à la même adresse ou que l'une d'entre elles contrôle l'exécution de l'autre). Aussi longtemps que les dépendances de données et de contrôle sont respectées, des optimisations variées et nombreuses sont réalisables autant du côté du compilateur que du côté du processeur. Le compilateur peut, par exemple, faire de l'allocation de registre, déplacer du code ou transformer des boucles. Le processeur, lui, peut exécuter plusieurs instructions en même temps, remplir son pipeline, contourner et rediriger le tampon d'écriture, employer des caches sans verrouillage, ... Toutes ces optimisations permettent de traiter plusieurs opérations simultanément et de ré-ordonnancer certaines opérations.

corriger la situation au niveau de la programmation ?

Au fil des années, plusieurs modèles de cohérence ont été introduits. Certains sont purement théorique et servent surtout à l'analyse d'algorithme. D'autres sont plus concrets. Ainsi, plusieurs langages de programmation et processeurs proposent leur modèle de cohérence qui fournissent dans la plupart des cas, une cohérence faible. Les langages C++ et Java introduisent des modèles de cohérence mémoire pour leur machine virtuelle sous-jacente. Les différents processeurs disponibles sur le marché (Intel, ARM, ...) offrent aussi chacun leur propre modèle de cohérence. De plus, il faut savoir que le modèle de cohérence du langage et celui du processeur ne concordent pas toujours. Dans ce cas, on doit toujours se préoccuper du modèle fourni par le langage.

Nous présentons ici plusieurs de ces modèles [109, 63, 37, 120, 150, 152, 14, 103] ainsi que, dans certains cas, leurs conséquences sur la conception d'algorithme sans verrous.

7.5.4 Cohérence sans synchronisation

Cohérence stricte ou atomique

Ce modèle de cohérence est celui qui impose le plus de contraintes sur l'ordre des opérations. Un système de mémoire partagé supporte un modèle de cohérence atomique, si la valeur retournée par une lecture à une adresse x est toujours celle de l'écriture la plus récente à cette même adresse, et ce peu importe la localisation des processus exécutant les opérations de lecture. Cela signifie que

Ré-ordonnacement des instructions

Le ré-ordonnancement des instructions, apparent ou réel, provient d'au moins une des trois sources suivantes :

- La mémoire cache ;

  Comme les modifications apportées aux données en cache ne sont pas reportées immédiatement en mémoire centrale, une apparence de ré-ordonnacement est probable.
- Le processeur [62, 135, 154, 8, 99, 146] ;

  Les processeurs modernes ont le potentiel d'exécuter dans le désordre certaines instructions, soit par le biais du super-scalaire, soit par un ré-ordonnancement explicite de celles-ci. En général, les instructions sont chargées (première étape du pipeline) dans l'ordre spécifié par le programme, mais l'éventualité qu'elles soient traitées dans le désordre existe et dépend de la disponibilité de leurs arguments. Considérons la suite d'instructions suivante :

  ```
  Load       A
  Load       B
  Increment  A
  Increment  B
  ```

  Il est possible que les deux instructions soient en attente en même temps pour l'obtention de leurs arguments. Si le chargement de B complète avant celui de A (dans le cas où B est en cache et pas A), alors l'incrément de B s'exécutera avant celui de A.
- Les compilateurs [62, 69, 98, 50] ;

  Les compilateurs, pour des raisons d'optimisation, ont le loisir de modifier explicitement l'ordre d'exécution des instructions.

les résultats de toutes les écritures sont instantanément visibles à tous les processus.

Cette définition suppose l'existence d'une horloge unique globale servant à ordonnancer totalement les opérations et à déterminer sans ambiguïté l'écriture la plus récente.

La cohérence stricte est supportée par un monoprocesseur et un monocœur. Implanter ce type de cohérence devient plus complexe sur un système multi-processeurs utilisant de la mémoire cache, voire impossible sur un système réparti vu l'absence d'une horloge globale.

Dans un système multi-processeurs pourvu de mémoire cache, la propagation de l'information vers la mémoire centrale et la mise à jour de toutes les caches ne sont pas instantanées. Assurer l'atomicité de ces opérations est possible mais ralentirait de façon significative les processeurs. C'est pourquoi la plupart des systèmes multiprocesseurs n'implantent pas une cohérence atomique.

Sur un système distribué, une lecture ou une écriture en mémoire éloignée exige un échange de messages avec l'ordinateur hébergeant la mémoire. Cet échange de message nécessite un temps considérable (comparativement au temps nécessaire à un processeur pour exécuter de nombreuses instructions) alors que, simultanément, d'autres processeurs ont l'opportunité d'accéder à cette même zone de mémoire. Implanter le modèle de mémoire atomique dans ce cas engendrerait des coûts bien supérieurs à ceux des systèmes multiprocesseurs.

Pour le plaisir, supposons un système de communication extrêmement efficace (latence nulle), des ordinateurs distants de trois mètres et un intervalle de demps entre deux accès de 2 nanosecondes. Dans ce contexte, à quelle vitesse le signal (message) doit-il voyager pour parcourir les trois mètres afin de devancer un éventuel second accès ? Selon Gehringer [37], le signal devrait atteindre 10 fois la vitesse de la lumière... On n'y est pas encore.

Heureusement, l'usage de ce type de cohérence est rarement nécessaire pour produire des programmes parallèles et d'autres modèles moins exigeants et répondant à la plupart des besoins sont disponibles.

Définition : Cohérence mémoire atomique (ou stricte)

Il y a cohérence atomique lorsque :

- toutes les écritures sont instantanément visibles à tous les processus ;
- toutes les lectures subséquentes retournent la nouvelle valeur de l'écriture et ce, peu importe le lieu d'exécution du processus et le délai entre l'initiation de la lecture et l'écriture.

Cohérence séquentielle

En classe, on a mentionné que l'on ne faut pas faire de suppositions quant aux vitesses relatives d'exécution des différents processus, ni à l'ordre dans lequel les opérations s'exécuteront ou s'entrelaceront. De fait, supposer que deux événements à l'intérieur d'un même processus se produisent suffisamment rapidement pour qu'aucun autre processus n'ait le temps de faire une action entre ceux-ci est source de problèmes. Il est toutefois possible de ré-ordonnancer les événements sans pour autant nuire à la bonne exécution d'un programme. À partir de ce dernier constat, la cohérence séquentielle a été formulée par Leslie lamport [54].

Un système de gestion mémoire respecte la cohérence séquentielle si tous les processus «voient» l'exécution de l'ensemble des opérations d'accès en mémoire exactement dans le même ordre (séquentiel) et que cet ordre respecte celui spécifié dans les différents programmes (ordre du programme). L'ordre exact dans lequel les opérations sont entrelacées n'a pas d'importance. Il faut imposer un

ordre total sur les écritures et s'assurer que les lectures locales aux processus retournent la dernière valeur écrite. En résumé, il existe un ordre global (total) sur tous les accès à la mémoire qui préserve l'ordre des instructions dans chancun des programmes.

À propos de l'ordre du programme

Soit les deux programmes suivants qui s'exécutent en parallèle (Initialement $X = 2$) :

```
        ...
(P1-1)  t1 = X;
(P1-2)  t1 = t1 + 4;
(P1-3)  X = t1;
        ...
```

Programme 7.3 – Processus $P1$

```
        ...
(P2-1)  t2 = X;
(P2-2)  t2 = t2 + 1;
(P2-3)  X = t2;
        ...
```

Programme 7.4 – Processus $P2$

Quelques exécutions qui respectent l'ordre des programmes :

- (P1-1) t1=X ;
- (P1-2) t1=t1+4 ;
- (P1-3) X=t1 ;
- (P2-1) t2=X ;
- (P2-2) t2=t2+1 ;
- (P2-3) X=t2 ;

X = 7

- (P1-1) t1=X ;
- (P2-1) t2=X ;
- (P1-2) t1=t1+4 ;
- (P2-2) t2=t2+1 ;
- (P1-3) X=t1 ;
- (P2-3) X=t2 ;

X = 3

- (P2-1) t2=X ;
- (P1-1) t1=X ;
- (P2-2) t2=t2+1 ;
- (P1-2) t1=t1+4 ;
- (P2-3) X=t2 ;
- (P1-3) X=t1 ;

X = 6

- (P2-1) t2=X ;
- (P2-2) t2=t2+1 ;
- (P1-1) t1=X ;
- (P1-2) t1=t1+4 ;
- (P2-3) X=t2 ;
- (P1-3) X=t1 ;

X = 6

La cohérence séquentielle diffère de la cohérence atomique de deux façons. Ainsi, selon la cohérence séquentielle :

- l'ordre temporel des événements n'a plus d'importance (plus besoin d'une horloge globale) ;
- la seule contrainte à respecter est d'imposer que tous les processeurs «voient» le même ordonnancement des opérations (et cela même si ce n'est pas l'ordre dans lequel elles se sont réellement exécutées dans le temps absolu).

L'exemple de la figure 7.39 illustre la différence entre ces deux types de cohérence. La séquence d'exécution 1 est acceptable selon le modèle de cohérence mémoire atomique, mais pas celle de l'exécution 2. En revanche, les deux séquences sont acceptables selon la cohérence séquentielle. Nous constatons donc que sur un système supportant la mémoire séquentiellement cohérente, d'exécuter un même programme deux fois peut produire des résultats différents (et être acceptable).

Même s'il est envisageable d'implanter la cohérence séquentielle, ce modèle génère de très mauvaises performances. Des modèles de cohérence moins contraignants et plus performants sont requis même s'ils ont un impact sur la conception d'algorithmes parallèles.

Modèle de cohérence causale (Casual Consistency Model)

On est à même de fournir un modèle de cohérence encore plus faible (et obtenir une meilleure performance). La prochaine étape pour affaiblir les contraintes de cohérence, consiste à faire une distinction entre les événements qui ont une connexion de cause à effet et ceux qui n'en ont aucune.

Définition : Cohérence séquentielle

Une mémoire est séquentiellement cohérente si le résultat de toutes les exécutions est le même que :

- si les opérations en mémoire de tous les processus sont exécutées dans un ordre séquentiel quelconque ;
- si les opérations de chacun des processus apparaissent dans l'ordre spécifié par le programme (ordre du programme).

Alors un ordre global (total) pour tous les accès à la mémoire qui préserve l'ordre séquentiel des programmes.
La cohérence séquentielle est «incluse» dans la cohérence atomique. Toute exécution séquentiellement atomique est automatiquement séquentiellement cohérente.

Temps ⟶			
P1	W(x)1		
P2		R(x)1	R(x)1
Initialement x = 0			

Exécution 1

Temps ⟶			
P1	W(x)1		
P2		R(x)0	R(x)1
Initialement x = 0			

Exécution 2

Figure 7.39 – Exemple d'exécutions

Deux événements ont une relation de cause à effet si l'un d'entre eux peut influencer l'autre. La figure 7.40 a illustre cette situation. Dans cet exemple, l'écriture de x a le potentiel d'influencer la valeur écrite dans y dû à la lecture (de x) entre les deux. Dans l'exemple de la figure 7.40 b, l'écriture de x n'a aucun effet sur celle de y, faute de lecture intermédiaire.

Temps ⟶			
P1	W(x)1		
P2		R(x)1	W(y)2
Initialement x = 0			

a W(x) peut influencer W(y)

Temps ⟶		
P1	W(x)1	
P2	W(y)2	
Initialement x = 0		

b W(x) n'influence pas W(y)

Figure 7.40 – Exemple d'exécutions

Les paires d'opérations suivantes ont un lien causal :

- Une lecture suivie d'une écriture
- Une écriture suivie par une lecture à la même adresse ;
- La fermeture transitive des deux types de relations précédentes.

Les opérations qui n'ont aucun lien causal sont dites concurrentes.

Une système de gestion mémoire supporte la cohérence causale si tous les processus observent dans le même ordre, les opérations (principalement les écritures) qui ont un lien causal. Les opé-

rations (écritures concurrentes) qui n'ont aucun lien causal peuvent être «vues» dans un ordre différent dans les processus impliqués.

Définition : Cohérence causale

Un modèle de cohérence est causal s'il garantit que les opérations d'écriture :

- causalement liées sont perçues par tous les processus dans le même ordre.
- non causalement liées peuvent être perçues dans des ordres différents, sur des processus différents

La figure 7.41 a présente un exemple d'exécution valide sur une mémoire à cohérence causale car il n'existe aucun lien entre les écritures des valeurs «2» et «3» dans x. Cette exécution est cependant illégale sur une mémoire séquentiellement cohérente. La figure 7.41 b présente un exemple d'exécution invalide sur une mémoire à cohérence causale dû au lien de cause à effet entre les deux écritures sur x.

Temps ⟶					
P1	W(x)1	W(x)3			
P2		R(x)1	W(x)2		
P3		R(x)1		R(x)3	R(x)2
P4		R(x)1		R(x)2	R(x)3
Initialement x = 0					

a Valide sur cohérence causale

Temps ⟶					
P1	W(x)1				
P2		R(x)1	W(x)2		
P3				R(x)2	R(x)1
P4				R(x)1	R(x)2
Initialement x = 0					

b Invalide sur une cohérence causale

Figure 7.41 – Exemple d'exécutions

Pour chacun de ces deux exemples :

1. Pourquoi l'exemple de la figure 7.41 a est-il invalide sous une mémoire séquentiellement cohérente ?
2. Sans l'opération $R(x)1$ dans le processus P_2, est-ce que la séquence de la figure 7.41 b deviendrait légale sous une mémoire causale ?

L'implantation d'une mémoire causale requiert la construction d'un graphe explicitant les relations de dépendance entre les opérations (graphe de dépendance).

Il est important de noter que le modèle de mémoire causale est un sous-ensemble du modèle de cohérence séquentielle.

Pipelined Random-Access Memory (PRAM)

La prochaine étape dans la création d'un modèle moins exigeant du côté cohérence est de n'exiger que seules les écritures issues d'un même processus soient perçues dans le même ordre.

Définition : Cohérence PRAM

Un système de gestion mémoire supporte la cohérence PRAM si toutes les opérations d'écriture provenant d'un même processus sont observées dans le même ordre par tous les autres processus, i.e. l'ordre dans lequel elles ont été exécutées. Les opérations d'écriture provenant de différents processus peuvent être perçues dans des ordres différents par tous les processus. On considère donc que les écritures émises par différents processus sont toutes concurrentes.

La cohérence PRAM a été nommée ainsi car les opérations d'écritures sont «vues» comme circulant dans un pipeline reliant chaque paire de processus. Le processus exécutant l'écriture n'a pas besoin de bloquer pour attendre la fin de l'exécution de l'opération.

L'exécution de la figure 7.41 b, qui violait la cohérence causale, respecte la cohérence PRAM. La figure 7.42 introduit une séquence d'exécution valide sur une mémoire PRAM mais qui produit un résultat plutôt contre-intuitif [103]. Si l'on retrace l'exécution de ces deux programmes, il semblerait logique qu'un seul des deux processus ou qu'aucun ne soit détruit. Mais selon le modèle de cohérence PRAM, il est possible d'obtenir une séquence dans laquelle les deux processus seront détruits. Pouvez-vous produire cette séquence ?

P1 : `a = 0;`		P2 : `b := 0;`
`...`		`...`
`a = 1;`		`b := 1;`
`if (b == 0) kill(p2);`		`if (a = 0) kill(p1);`

Figure 7.42 – Exemple d'exécutions

Il est important de noter que le modèle de mémoire PRAM est un sous-ensemble du modèle de cohérence causale.

Cohérence cache (ou par objet)

Un système respecte un modèle de cohérence objet et de cohérence cache, si les écritures sur un même objet (dans une base de données) ou en un même emplacement en mémoire sont perçues dans un même ordre par tous les processus. En revanche, il n'est pas nécessaire que les écritures à des adresses distinctes soient «vues» dans le même ordre par tous. C'est donc une cohérence « naturelle » lorsqu'on utilise un protocole de cohérence de cache sur les multiprocesseurs ou multi-cœurs.

Définition : Cohérence cache

Un système supporte la cohérence cache (ou objet) si tous les accès à chaque objet sont séquentiellement cohérents.

Il est important de noter que le modèle de cohérence cache ne se compare pas au modèle de cohérence PRAM.

Cohérence de processeur (Processor consistency) [120]

La cohérence de processeur combine les types de cohérence PRAM et cache. Ainsi, un système implante ce type de cohérence si :

- les opérations d'écriture émises par un processus sont perçues par tous dans le même ordre que les instructions du programme (PRAM) ;
- les écritures à la même adresse sont «vues» dans le même ordre par tous les processus (cohérence cache).

Définition : Cohérence de processeur

Un système supporte la cohérence de processeur si elle implante à la fois la cohérence PRAM et la cohérence cache.

Attention : Autre définition pour la cohérence de processeur

Singhal [93, 41] définit la cohérence de processeur comme étant identique au modèle de mémoire PRAM.

La cohérence de processeur est moins contraignante que le modèle causal mais plus restrictive que le modèle PRAM. Les exemples suivants présentent des situations qui permettent de distinguer ces modèles.

Temps ⟶				
P1	W(x)1	W(x)3		
P2			R(x)1	R(x)3
P3	W(y)1	W(y)2		
P4			R(y)1	R(y)2

Initialement x = 0

a Cohérent selon le modèle processeur.

Temps ⟶				
P1	W(x)1	W(x)3		
P2			R(x)3	R(x)1
P3	W(y)1	W(y)2		
P4			R(y)2	R(y)1

Initialement x = 0

b Non cohérent selon le modèle processeur

Figure 7.43 – Exemple d'exécutions

Temps ⟶					
P1	W(x)1				
P2		R(x)1	W(x)2		
P3				R(x)1	W(x)2
P4				R(x)2	R(x)1

Initialement x = 0

Figure 7.44 – Non cohérent causal mais cohérent processeur

Cohérence lente (slow memory) [152, 63]

La cohérence lente constitue modèle PRAM affaibli car il ne considère que la cohérence par processus uniquement basé sur la localisation. Ce modèle requiert donc que tous les processeurs s'accordent sur l'ordre des écritures à chaque localisation mais seulement par un seul processus. De

Temps ⟶						
P1	W(x)2	W(y)4	W(x)3	W(y)1		
P2			R(y)4	R(x)2	R(y)1	R(x)3
Initialement x = 0						

Figure 7.45 – Exemple d'exécution non cohérente processeur mais cohérente cache

plus, les écritures locales sont visibles immédiatement. Ce modèle est probablement l'un des plus faibles qui existent. La figure 7.46 introduit un exemple d'exécution qui respecte ce modèle mais qui est invalide dans le modèle PRAM.

Temps ⟶				
P1	W(x)1	W(y)1		
P2			R(y)1	R(x)0
Initialement x = 0				

Figure 7.46 – Exemple d'exécution non cohérente processeur mais cohérente cache

Selon Mosberger[63], ce modèle ne s'avère guère utile en pratique.

Cohérence à terme (eventual consistency)[150, 153]

Le modèle de cohérence à terme, aussi appelé «optimistic replication», est un modèle couramment utilisé dans les systèmes distribués pour assurer une haute disponibilité. Il exige, si aucune nouvelle mise à jour n'est initiée, que toutes les lectures retournent éventuellement la dernière (et la même) valeur écrite. Il n'y a toutefois aucun temps limite ni aucune contrainte d'ordre sur les opérations d'écritures. C'est le modèle de cohérence le plus faible employé en pratique.

7.5.5 Modèles de cohérence avec synchronisation [103, 52, 20]

Les modèles précédents ne font aucune distinction entre les différents types de données accédées. Ces modèles tentent donc de maintenir la cohérence à tout instant sur toutes les données, ce qui s'avère parfois inutile. Ainsi les modèles de cohérence PRAM et processeur fournissent une cohérence plus forte que requise par plusieurs programmes car ceux-ci ne requièrent pas toujours de percevoir toutes les écritures provenant d'un même processeur dans le même ordre, ni même, d'ailleurs, de toutes les «voir».

Par exemple, soit un processus P_1 qui itère sur une condition en section critique, lisant et modifiant des variables. Les autres processus ne sont pas sensés accéder à ces variables tant que P_1 demeure en section critique. Sous le modèle PRAM, le gestionnaire de la mémoire ignore que les autres processus ne veulent pas accéder à ces variables. Il doit donc transmettre constamment toutes les écritures à tous les processus et s'assurer qu'elles arrivent dans l'ordre (même si cela est inutile tant que P_1 ne sort pas de la section critique).

De ce constat, vient l'idée d'assouplir le modèle de cohérence et de forcer la synchronisation uniquement lorsqu'elle est nécessaire et ce, à l'aide de variables et d'opérations de synchronisation.

Dans ces nouveaux modèles, on distingue les données normales des données servant à la synchronisation. Des instructions de synchronisation, appliquées uniquement aux données de synchronisation, permettent de garantir temporairement une cohérence plus forte. Ces instructions sont généralement des « barrières » pour assurer à tous les processus la visibilité aux accès précédents. Cela permet d'être plus spécifique lors de la synchronisation et ainsi de réduire les coûts de cette dernière.

Les sections suivantes présentent différents modèles de cohérence à base de synchronisation.

Cohérence faible (Weak consistency)

Selon le modèle de cohérence dit «faible» (Weak consistency), la propagation des mises à jour des données «normales» est effectuée seulement lorsque des variables de synchronisation sont utilisées. Pour assurer la cohérence des données, l'accès aux données normales est toujours encadré par des opérations manipulant des données de synchronisation (une «section critique»). Les accès aux variables de synchronisation sont séquentiellement cohérents ;

Selon ce modèle, il n'est pas nécessaire de propager immédiatement les résultats des accès se produisant dans une section critique. Il est suffisant d'attendre la sortie de la section critique du processus effectuant les mises à jour et, seulement alors, de s'assurer que les résultats sont propagés à tout le système. Pour ce faire, on doit de bloquer les accès à ces variables tant que la propagation n'est pas terminée. De même, lorsque l'on désire entrer en section critique, il faut s'assurer que toutes les écritures antérieures soient entièrement terminées.

Définition : Cohérence faible

Un système implante une cohérence dite faible si toutes les opérations aux variables de synchronisation sont séquentiellement cohérentes. Ainsi, avant qu'une opération de synchronisation ne soit exécutée, toutes les opérations sur des données régulières doivent être complétées. De même, avant qu'une opération sur une donnée régulière puisse être exécutée, toutes les opérations de synchronisation antérieures doivent être complétées. Cette façon de faire laisse à la personne en charge du développement, la responsabilité de la cohérence des données car la ne mémoire sera cohérente seulement qu'après une opération de synchronisation.

Un système implante la cohérence faible si et seulement si :

- Les accès aux variables de synchronisation sont séquentiellement cohérents ;
- Aucun accès aux variables de synchronisation n'est permis tant que les écritures antérieures ne sont pas entièrement complétées ;
- Aucun accès aux variables (lecture ou écriture) n'est permis tant que les accès antérieurs aux variables de synchronisation ne sont pas terminés.

Ainsi, en ayant recours à la synchronisation avant de lire une donnée, un processus s'assure d'obtenir la valeur la plus récente.

Il est important de noter que ce modèle ne permet pas l'exécution de plus d'une section critique à la fois et cela même si elles sont disjointes (aucune variable partagée).

Ce modèle laisse la lourde responsabilité à la personne en charge de la programmation de choisir les variables de synchronisation.

La figure 7.47 introduit deux séquences d'exécutions légales sur une mémoire à cohérence faible. En effet, aucune contrainte n'est nécessaire avant la synchronisation (S). Après la synchronisation, la valeur de x est cohérente (7.47 b). La figure 7.48 présente un séquence d'exécutions illégales car

la valeur disponible après la synchronisation ne respecte pas l'ordre d'exécution.

Temps ⟶					
P1	W(x)1	W(x)2			S
P2			R(x)2	R(x)1	S
P3			R(x)1	R(x)2	S
Initialement x = 0					

a Exemple 1

Temps ⟶						
P1	W(x)1	W(x)2			S	
P2			R(x)0	R(x)2	S	R(x)2
P3				R(x)1	S	R(x)2
Initialement x = 0						

b Exemple 2

Figure 7.47 – Exemple d'exécution respectant la cohérence faible

Temps ⟶				
P1	W(x)1	W(x)2	S	
P2			S	R(x)1
Initialement x = 0				

Figure 7.48 – Exemple d'exécutions ne respectant pas la cohérence faible

Cohérence à la libération (Release Consistency)

La cohérence faible ne fait aucune distinction entre les opérations d'entrée ou de sortie de la section critique. Ainsi, sur ces deux opérations, un système supportant la cohérence faible doit accomplir toutes les actions nécessaires à la synchronisation. S'il était possible de distinguer ces deux opérations, il serait possible de réduire le nombre d'opérations de synchronisation requises.

La cohérence à la libération a pour objectif de faire cette distinction. Elle fournit donc deux opérations :

- l'acquisition qui sert à entrer en section critique ;

  À cette étape, le système de gestion de la mémoire s'assure que toutes les mises à jour provenant de d'autres processus (ou processeurs) sont localement terminées. On vérifie que les copies locales des données sont cohérentes avec leurs versions distantes (i.e. qu'elles sont à jour).

  Il n'y a aucune garantie que les changements locaux seront propagés au moment de l'acquisition.

- la libération qui sert à sortir de la section critique.

  À cette étape, les mises à jour initiées localement sur des variables partagées sont propagées aux autres processus ou sites.

  Elle ne garantit pas que les modifications faites par les autres processeurs seront prises en compte.

De plus, avec ce modèle, il est possible d'utiliser plusieurs variables de synchronisation, ce qui permet à plusieurs sections critiques de s'exécuter simultanément.

Définition : Cohérence à la libération

La cohérence à la libération est similaire à la cohérence faible sauf en ce qui concerne les opérations de synchronisation qui n'ont qu'à respecter la cohérence de processeur l'un par rapport à l'autre. Les opérations de synchronisation sont divisées en deux parties, l'acquisition et la libération, et toutes les opérations d'acquisition non complétées sont tenues de l'être avant la fin de l'opération de libération. Les dépendances locales sur un processeur doivent être respectées.
La cohérence à la libération est un assouplissement des contraintes par rapport à la cohérence faible sans pour autant perdre significativement de la cohérence.

Un système implantant la cohérence à la libération obéit aux règles suivantes :

- Avant d'accéder à une variable ne servant pas à la synchronisation, toutes les opérations d'acquisition (opération de synchronisation) faites par le processus doivent être complétées.
- Avant de libérer un section critique, toutes les opérations de lecture et écriture antérieures faites par le processus doivent être complétées.
- Les opérations d'acquisition et de libération doivent respecter la cohérence de processeur (PRAM).

Si ces conditions sont satisfaites et que les processus manipulent les opérations de synchronisation correctement, les résultats des exécutions conséquentes seront toujours les mêmes que s'ils étaient produits sur une mémoire séquentiellement cohérente.

La figure 7.49 présente une séquence d'exécution valide sur une mémoire implantant la cohérence à la libération. Dans cet exemple, le processus $P3$ n'obtient pas la valeur la plus récente de x car il n'effectue aucune synchronisation.

Temps ⟶								
P1	A(V)	W(x)1	W(x)2	L(V)				
P2					A(V)	R(x)2	L(V)	
P3								R(x)1

Initialement x = 0

Figure 7.49 – Exemple d'exécution respectant la cohérence à la libération

Pour mieux comprendre l'effet des opérations d'acquisition et de libération, la figure 7.50 introduit un cas d'implantation faisant appel à un gestionnaire centralisé [103]. Évidemment, des implantations sans aucun gestionnaire central existent aussi.

Cohérence à l'entrée (Entry Consistency)

La cohérence à l'entrée est similaire à la cohérence à la libération. Comme pour cette dernière, la personne en charge du développement doit employer des opérations de synchronisation distinctes (acquisition et libération) à l'entrée et à la sortie des sections critiques. Toutefois, la cohérence à l'entrée exige que toutes les variables dites normales soient associées à une certaine variable de synchronisation. Lors de l'opération d'acquisition sur une variable de synchronisation, seules les variables «normales» qui lui sont associées sont mises à jour et conservées cohérentes.

Exemple d'implantation de la cohérence à la libération [103]

Lors de l'acquisition, un processus envoie un message à un gestionnaire de la synchronisation demandant le verrouillage d'une certaine région critique (un verrou précis). S'il n'y a aucune compétition, le verrouillage est accordé et l'acquisition complétée.
Le processus lit et écrit alors les variables partagées associées au verrou. Aucune des modifications ne sera propagée aux autre sites.
Lors de la libération, les données modifiées sont communiquées aux autres sites qui les utilisent. À leur tour, chacun d'eux confirme détenir ces nouvelles valeurs. Uniquement au moment où tous les accusés de réception sont reçus, le processus informe le gestionnaire de la synchronisation que le verrou est maintenant libéré.

Figure 7.50 – Exemple d'implantation de la cohérence de libération

Un système de mémoire implantant la cohérence à l'entrée doit respecter les trois conditions suivantes :

- L'acquisition d'une variable de synchronisation n'est permise que lorsque toutes les modifications sur les variables normales associées sont complétées ;
- Avant le verrouillage d'une variable de synchronisation, aucun autre processus ne doit manipuler la variable ;
- Après le déverrouillage d'une variable de synchronisation, les prochains accès devront respecter les deux étapes précédentes.

Autres modèles

D'autres modèles avec synchronisation existent, en particulier :

- la cohérence de libération paresseuse (lasy release consistency) [103] ;
- la cohérence de portée (scope consistency) [46] ;

7.5.6 Modèles de cohérence sur les multiprocesseurs [5, 150, 152]

Les modèles que nous venons de présenter s'appliquent principalement aux systèmes de gestion de mémoire distribué. D'autres modèles spécialisés pour les multiprocesseurs existent et servent surtout à assurer la cohérence des données entre les différentes mémoires cache et la mémoire centrale (partagée par tous). Ces modèles ont la particularité d'être implantés au niveau matériel.

Dans les systèmes multiprocesseurs, les modèles de cohérence garantissent, pour des raisons de performance, très peu de cohérence. Cette cohérence «relâchée» rend possible certaines optimisations, comme l'exécution d'instructions hors séquence. Ces optimisations permettent en particulier de poursuivre l'exécution d'un programme sans attendre que le résultat des écritures antérieures ne soient rendues visibles à tous les processeurs. Ces modèles sont similaires à la cohérence faible en ce sens qu'ils se fient à des instructions spéciales pour assurer une certaine synchronisation. Ils sont aussi particulier du fait d'une implantation matérielle basée sur des tampons d'écriture et des protocoles de cohérence de cache.

Principes de base

L'un des principes de base de la cohérence mémoire sur multiprocesseurs est de cacher la latence des écritures. Cette latence correspond au temps pris par une écriture pour se propager vers la mémoire centrale et vers toutes les autres copies de la mémoire cache. Cette latence est extrêmement longue par rapport au temps de traitement d'une instruction. Il en résulte que, dans de nombreux cas, certaines opérations (telle une lecture) soient exécutées avant la fin de l'exécution d'une écriture malgré qu'elles apparaissent après celle-ci dans l'ordre du programme. Dans les faits, plusieurs applications fonctionnent très bien même si l'ordre du programme n'est pas respecté. Ainsi, la mémoire apparaît tout de même comme implantant la cohérence séquentielle.

Rappelons que la cohérence séquentielle impose deux contraintes : le respect de l'ordre des instructions dans le programme et l'atomicité des écritures. L'ordre du programme garantit que chaque processus exécute les instructions d'accès à la mémoire selon l'ordre du programme et l'atomicité, elle, garantit que les demandes en mémoire sont servies dans l'ordre d'une seule file de type FIFO.

Plusieurs modèles de cohérence «relâchés» sont obtenus en allégeant l'une de ces deux contraintes :

1. Ordre du programme ;

   Les allègements possibles consistent à ne pas respecter l'ordre du programme pour une ou plusieurs des paires d'instructions suivantes n'accédant pas à la même adresse :

   - écriture → lecture ;
   - écriture → écriture ;
   - lecture → lecture ;
   - lecture → écriture.

2. Atomicité des écritures

   Généralement, l'atomicité assure qu'une lecture retourne une valeur seulement si elle est déjà visible pour tous (l'écriture est complétée). Il y a deux allègements possibles :

   - permettre à un processus de lire le résultat d'une des ses écritures antérieures avant qu'elle ne soit complétée pour les autres processus (Read oWn's Early - RWE) .
   - permettre à tous les processus de lire le résultat d'une écriture avant qu'elle ne soit complétée pour tous (Read Other's Early - ROE).

Alléger la contrainte d'atomicité : écriture ⟶ lecture

L'une des premières formes d'allègement est de permettre l'exécution d'une lecture avant même qu'une écriture antérieure ne termine sur un processeur particulier. Cette approche permet de cacher la latence des écritures.

Il existe trois modèles basés sur cet allègement qui assouplissent différemment les contraintes d'atomicité :

1. Le modèle IBM 370 (mainframe fabriqué pas IBM dans les années 1970-1980 [155])

   Selon le modèle de cohérence mémoire du IBM 370, une lecture peut être complétée avant une écriture antérieure à une adresse différente. Il est toutefois interdit de retourner la valeur d'une écriture si elle n'est pas déjà complétée (visible à tous les processeurs). L'atomicité est préservée.

Dans ce modèle, il est possible d'exécuter une écriture suivie par une lecture dans le désordre (sans respecter l'ordre du programme), sauf en quelques exceptions. L'ordre doit être respectée si les opérations accèdent la même adresse, si l'une des opérations en est une de sérialisation (synchronisation - barrière) ou s'il y a une opération de sérialisation entre les deux opérations.

2. Les modèle TSO du SPARC V8 (Total store ordering)

   Le modèle TSO allège partiellement l'atomicité requise par le modèle du IBM 370. Il permet à une lecture de retourner une valeur écrite à partir du même processeur avant qu'elle ne soit propagée aux autres processeurs. De façon similaire au modèle précédent, une lecture ne peut retourner le résultat d'une écriture externe au processeur courant sans être complétée (allègement de type RWE).

   Ce modèle permet donc partiellement à deux opérations visant la même adresse de s'exécuter (du moins en apparence) dans le désordre. Il s'implante principalement grâce à des tampons d'écriture situés entre le processeur et la cache de niveau L1 comme l'indique la figure 7.51. Dans cette architecture, chaque fil se voit attribuer un tampon d'écriture. La valeur d'une écriture y est copiée et, ceci fait, le fil poursuit son exécution (sans attendre que le résultat soit transmis en mémoire centrale). Une lecture subséquente obtient la valeur la plus récente écrite à la même adresse directement du tampon si elle est disponible.

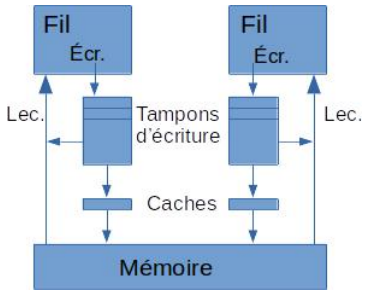

Figure 7.51 – Le modèle TSO du SPARC V8

Ce modèle admet donc qu'une lecture (à partir d'un autre processeur que celui qui a fait l'écriture) retourne une valeur qui n'est plus à jour en terme d'ordre de programme. Cependant, une fois l'écriture complétée, elle l'est pour tous en même temps.

3. Les modèle TSO du x86 (Total store ordering)

   Ce modèle possède plusieurs définitions, mais à la base il est relativement similaire au modèle du SPARC V8. La principale différence entre les deux provient de l'omission de certaines instructions et de l'inclusion de d'autres. En fait, le tampon d'écriture utilise plusieurs états et verrous afin de déterminer si une valeur peut être lue ou modifiée.

4. La cohérence de processeur

   Ce modèle allège une contrainte supplémentaire reliée à l'atomicité par rapport au modèle TSO, en ce sens qu'une lecture (peu importe sa source) peut retourner la valeur d'une écriture même si elle n'est pas complétée (donc possiblement une valeur non à jour, modèle ROE). Dans ce modèle, à un moment donné, il est admissible qu'une écriture soit complétée pour un processeur sans qu'elle le soit pour un autre.

L'exemple de la figure 7.52 met en évidence l'effet produit en levant la contrainte «écriture $\longrightarrow$ lecture». Une mémoire respectant la cohérence séquentielle n'autorisera pas l'impression de $A = 0$ et $B = 0$. Toutefois, l'impression de ces deux valeurs est possible pour tous les modèles présentant ce premier allègement (IBM370, TSO et PC).

→ Au départ : A = B = 0		
P1 : A = 1 Print B		P2 : B = 1 Print A

Figure 7.52 – Exemple 1 : Allègement écriture $\longrightarrow$ lecture

L'exemple de la figure 7.53 permet de mieux comprendre la distinction entre ces différents modèles. Le résultat obtenu à la fin de l'exécution ne peut être obtenu avec le modèle du IBM 370. En effet, la lecture de A ne s'exécutera pas avant que l'écriture sur A ne soit complétée sur ce même processeur. Comme l'ordre des écritures est maintenue dans ce modèle, les lectures des $flag1$ et $flag2$ ne s'exécuteront pas avant que les écritures sur ces mêmes variables ne soient elles aussi complétées. Ce même résultat est toutefois possible avec les modèles TSO et PC car ils permettent aux lectures des $flag1$ et $flag2$ de se faire avant que les écritures sur le même processeur ne soient complétées.

→ Au départ : A = flag1 = flag2 = 0		
P1 : flag1 = 1 A = 1 reg1 = A reg2 = flag2		P2 : flag2 = 1 A = 2 reg3 = A reg4 = flag1
→ À la fin : reg1 = 1 ; reg3 = 2, reg2 = reg4 = 0		

Figure 7.53 – Exemple 2 : Allègement écriture $\longrightarrow$ lecture

Dans l'exemple de la figure 7.54, le résultat présenté ne s'obtient que par le modèle PC car celui-ci permet au processus $P2$ de retourner la valeur de A avant qu'elle ne soit visible à $P3$. Cette situation ne peut se produire avec les autres modèles puisque l'écriture initiée par $P1$ doit être complétée pour tous les autres processeurs en même temps, soit pour $P2$ et $P3$.

→ Au départ : A = B = 0		
P1 : A = 1	P2 : ... if (A == 1) B = 1	P3 : ... ... if (B == 1) reg1 = A
→ À la fin : B = 1, reg1 = 0		

Figure 7.54 – Exemple 3 : Allègement écriture $\longrightarrow$ lecture

Pour fournir la cohérence séquentielle dans ces modèles, on fait appel à des instructions de type

barrière. Le IBM370 fournit des opérations de sérialisation spéciales à insérer entre deux énoncés d'accès à la mémoire. Sur les modèles TSO et PC, les personnes en charge de développer des programmes doivent utiliser des opérations de synchronisation (du style «`read-modify-write`») pour s'assurer de maintenir l'ordre du programme.

Alléger la contrainte : écriture $\longrightarrow$ lecture et écriture $\longrightarrow$ écriture

Ce second allègement des contraintes d'ordonnancement est plus permissif que le premier car il autorise des opérations d'écriture à des adresses différentes à s'exécuter dans le désordre. Le modèle PSO (Partial Store Ordering) du processeur SPARC V8 constitue l'unique implantation de ce modèle.

Ce modèle admet l'exécution simultanée (dans un pipeline) des écritures à des adresses distinctes à partir du même processeur. Autrement, PSO est similaire à TSO en terme d'exigence d'atomicité. Il permet à un processeur de lire la valeur des ses propres écritures mais empêche les processeurs de lire le résultat d'une écriture d'un autre processeur avant qu'elle ne soit visible à tous.

Les résultats produits dans les exemples des figures 7.53 et 7.54 sont autorisés selon le modèle PSO.

Pour conserver l'ordre du programme entre deux écritures, il est nécessaire de recourir à une opération spéciale appelée STBAR (une forme de barrière). Cette dernière s'insère généralement dans le tampon d'écriture (lorsque celui-ci est implanté telle une file de type FIFO) et retarde toutes les écritures qui sont emmagasinées à sa suite, aussi longtemps que celles qui la précèdent (l'instruction STBAR) ne soient complétées. Un compteur sert généralement à déterminer si toutes les écritures sont terminées. Une écriture en mémoire (précédant le STBAR) incrémente le compteur et un accusé de réception sur une écriture (fin de l'écriture) le décrémente. Lorsque le compteur devient 0, les écritures à la suite du STBAR débutent.

PSO conserve l'ordre du programme «écriture $\rightarrow$ lecture» et garantit l'atomicité des écritures (partiellement avec RWE).

Alléger les contraintes : lecture $\longrightarrow$ lecture et lecture $\longrightarrow$ écriture

Les modèles de cohérence matériels précédents, TSO, PC et PSO, ne possèdent pas la flexibilité nécessaire pour permettre des optimisations intéressantes au niveau des compilateurs. Un modèle plus flexible est requis.

Ainsi, ce dernier allègement, ajouté aux précédents, lève toutes les contraintes sur l'ordre séquentiel des instructions d'un programme à des adresses différentes. Une lecture ou une écriture pourra dorénavant être ré-ordonnancée par rapport aux lectures ou écritures qui la suivent (toujours à différentes adresses).

Pour assurer l'ordonnancement des instructions, des énoncés spéciaux sont requis.

Différents modèles proposent ce type d'allègement :

- La cohérence faible (principalement la cohérence à la libération - «Release Consistency») que nous avons déjà abordée ;

  Comme déjà mentionné, ce modèle exige que l'accès aux données soit toujours encadré par des instructions de synchronisation. Les deux modèles les plus populaires sont :

  – la cohérence à la libération, appelée RC_{SC}, selon laquelle les opérations de synchronisation respectent la cohérence séquentielle et,

  - la cohérence RC_{PC} selon laquelle les opérations de synchronisation respectent seulement la cohérence de processeur.

- le modèle RMO (Relax Memory Order) implanté par le SPARC V9 ;
  Ce modèle est accompagnée d'une instruction appelée «MEMBAR» qui se configure pour ordonnancer les lectures ou les écritures qui la précèdent par rapport aux opérations qui la suivent.
- Le modèle proposé par le processeur ALPHA (Digital) ;
  Ce processeur fournit deux instructions de synchronisation de type barrière :
  - la barrière mémoire (Memory Barrier - MB)
    Cette instruction assure l'ordre du programme pour toutes les instructions qui précèdent la barrière avec celles qui la suivent.
  - la barrière d'écriture (Write Memory Barrier - WMB)
    Cette instruction garantie l'ordre du programme pour toutes les instructions d'écriture qui précèdent la barrière avec les opérations d'écriture qui la suivent.
- Le modèle proposé par le PowerPC de IBM ;
  Ce modèle requiert une seule opération de barrière appelé SYNC. Cette opération ressemble à la barrière mémoire (MB) du processeur ALPHA à quelques nuances près expliquées par Adve [5].
- Le modèle proposé par l'architecture ARMv7.
  Le modèle de cohérence adopté par le processeur ARM [95, 68] semble équivalent en terme d'allègement à celui proposé par le processeur PowerPC. Il introduit plusieurs instructions de barrière pour assurer la synchronisation.
- Le modèle du RISC-V
  Le processeur RISC-V fait appel à un modèle de cohérence appelé RVWMO (Risc-V Weak Memory Order). Ce modèle est similaire à la cohérence à la libération. Pour assurer la synchronisation, le processeur implante plusieurs instructions de type barrière. De plus, les lectures et les écritures se paramétrisent afin de fournir une acquisition ou une libération respectivement.
- Le modèle proposé par l'architecture ARMv8.
  Le modèle introduit pour le ARMv8 ressemble à celui du RISC-V. Les lectures et les écritures s'annotent avec des opérations de synchronisation, respectivement des acquisitions et des libérations.

Dans tous ces modèles il est possible de ré-ordonnancer toutes les opérations en mémoire si elles ciblent des adresses distinctes.

De plus, certains allègements sur l'atomicité sont également envisageables. Ainsi, les modèles du SPARC et du PowerPC autorisent le ré-ordonnancement des lectures à la même adresse. Les modèles des processeurs ALPHA et SPARC (RMO) permettent à une lecture de retourner des valeurs dont l'écriture n'est pas entièrement terminée (ROE).

7.5.7 Mémoire transactionnelle

Le modèle de mémoire transactionnelle fournit à la fois la cohérence de la mémoire et la cohérence du cache. Comme nous l'avons déjà vu, une transaction est une séquence d'opérations exécutée par

un processus qui transforme les données d'un état cohérent à un autre. Une transaction est soit validée lorsqu'il n'y a pas de conflit, soit abandonnée. Dans le cas d'un engagement, toutes les modifications sont visibles pour tous les autres processus lorsque la transaction est terminée, tandis que dans le cas d'un abandon, toutes les modifications sont supprimées. Ce modèle est plus facile à manipuler et offre des performances plus élevées qu'un modèle de cohérence séquentiel.

7.5.8 Autres modèles

Plusieurs autres modèles existent [7, 31, 152, 150, 103, 63]. Sans les présenter en détails, voici quelques exemples de modèles :

- Cohérence à terme (modèle sans synchronisation)
  Le modèle de cohérence à terme (eventual consistency) n'exige pas de mise à jour simultanée mais requiert que :
  - les écritures d'un processus soient éventuellement vues par les autres mais sans date limite, ni contrainte d'ordre ;
  - lorsque les mises à jour s'arrêtent, tous les processus observent le même état.

  C'est le modèle de cohérence le plus faible utilisé en pratique. On le retrouve dans certaines bases de données distribuées.
- Cohérence Delta [7, 31, 152] ;
  La cohérence delta requiert que la propagation d'une opération d'écriture se fasse à l'intérieur d'une période de temps fixe. L'ensemble du système est donc synchronisé à intervalles de temps régulier. Autrement dit, le résultat d'une lecture mémoire est généralement cohérent à l'exception d'une courte période de temps, celle où la synchronisation n'a pas encore été effectuée. Donc, si un espace mémoire a été modifié, pendant une courte période les accès en lecture seront incohérents. Il faudra attendre la synchronisation pour retrouver des informations cohérentes.
- Cohérence d'embranchement (Fork consistency) [7, 31, 152] ;
  Ce modèle introduit le concept de processus de confiance. Ainsi une écriture faite par un processus ne faisant pas partie du groupe de confiance pourrait ne jamais être visible pour les autres processeurs. De même, les écritures faites par les autres processus pourraient ne jamais lui être visibles. Il y a donc création de plusieurs vues de l'information (embranchements ou fork).
- Sérialisation [144] ;
  Ce modèle concerne principalement les bases de données ou les environnements ayant recourt au concept de transaction. Un suite de transactions est sérialisable si le résultat dans la base de données est le même que celui obtenu dans la cas où toutes les transactions auraient été exécutées en séquence.
- Linéarisation [132, 7, 31, 152]
  Un ensemble d'opérations parallèles est «linéarisable» s'il consiste en une liste d'appels (demandes) et de réponses à des événements qui, une fois complétée par l'ajout des réponses manquantes, peut être ré-écrite comme un historique séquentiel. Ce dernier doit nécessairement être un sous-ensemble de la liste originale.

Informellement cela signifie que la liste originale d'événements est linéarisable si et seulement si tous les appels sont linéarisables mais que certaines réponses sont absentes (n'ont pas encore été retournées).

Le comportement de ce modèle est similaire à celui de la cohérence séquentielle sauf que l'ordre des opérations est déterminé par l'ensemble des processus basé sur une notion de temps (horloge logique par exemple).

La linéarisabilité est un concept plus général que la sérialisabilité dans les systèmes distribués et elle ressemble beaucoup à la cohérence stricte (étant donné la notion de temps qui est présente).

- Cohérence par champs de vecteurs [7, 31, 152] ;

  Ce modèle a été initialement créé pour la gestion de copies multiples de données dans les environnements de jeu vidéo afin de minimiser l'utilisation de la bande passante et le temps d'accès. Le principe est que, même si une personne jouant à un jeu a besoin d'informations sur l'entièreté du jeu, elle nécessite davantage de connaître plus rapidement et avec une plus grande précision, les informations concernant les éléments du jeu dans son environnement immédiat. Le modèle permet donc un niveau d'incohérence qui dépend d'une notion de distance. Ainsi, le modèle de cohérence est sélectionné, renforcé ou affaibli, selon la position du joueur dans le jeu.

- Cohérence locale [7, 31, 152] ;

  La cohérence locale est considérée comme le modèle de cohérence le plus faible. Il y est seulement exigé que les opérations locales, écritures ou lectures, apparaissent localement dans l'ordre séquentiel défini par le programme. Il n'y existe aucune contrainte d'ordonnancement pour les processeurs concernant les écritures distantes.

7.5.9 Sommaire et comparaison

En résumé, la cohérence stricte (ou atomique) est pratiquement impossible à implanter. La cohérence séquentielle elle, l'est mais s'avère trop coûteuse. Les multiples modèles présentés allègent ensuite les contraintes de différentes façons.

Les modèles n'employant pas la synchronisation sont résumés dans le tableau 7.1. La figure 7.55 (inspiré de [6]) présente la hiérarchie existante entre tous ces modèles de cohérence mémoire. Le tableau 7.2 résume les modèles de cohérence avec synchronisation. Le tableau 7.3 résume les allègements autorisés par les modèles ou les architectures.

7.5.10 Gestion d'espaces d'adresses de 64 bits [87, 110, 158]

Dans les systèmes actuels, chaque processus possède son propre espace d'adresses qui varie entre 2^{32} et 2^{64} octets. Aucun processus ne peut accéder à l'espace d'adresses d'un autre. Ils sont isolés par les mécanismes d'adressage logique et virtuel.

Dans les années 90, avec l'apparition des espaces d'adresses de 64 bits, certains systèmes ont voulu briser cette approche en proposant un espace d'adresses unique pour tous. C'est ce qu'on appelle les «Single Address Space Operating System» (SASOS). Selon ce type de système, une adresse pointe au même endroit pour tous les processus. On se retrouve donc dans un énorme système multi-fils!!!

Type de cohérence	Description
Stricte	Tous les processus voient les accès dans l'ordre dans lequel ils se sont exécutés selon une horloge globale absolue.
Sequentielle	Tous les processus voient les accès aux données partagées dans le même ordre.
Causale	Tous les processus voient les accès ayant un lien de cause à effet (causalité) dans le même ordre.
Pram	Tous les processus voient les écritures provenant de chacun des processeurs dans l'ordre dans lequel ils ont été initiés. Les écritures provenant de processus distincts ne sont pas ordonnés et peuvent donc être perçus dans des ordres différents.

Table 7.1 – Modèles de cohérence sans synchronisation

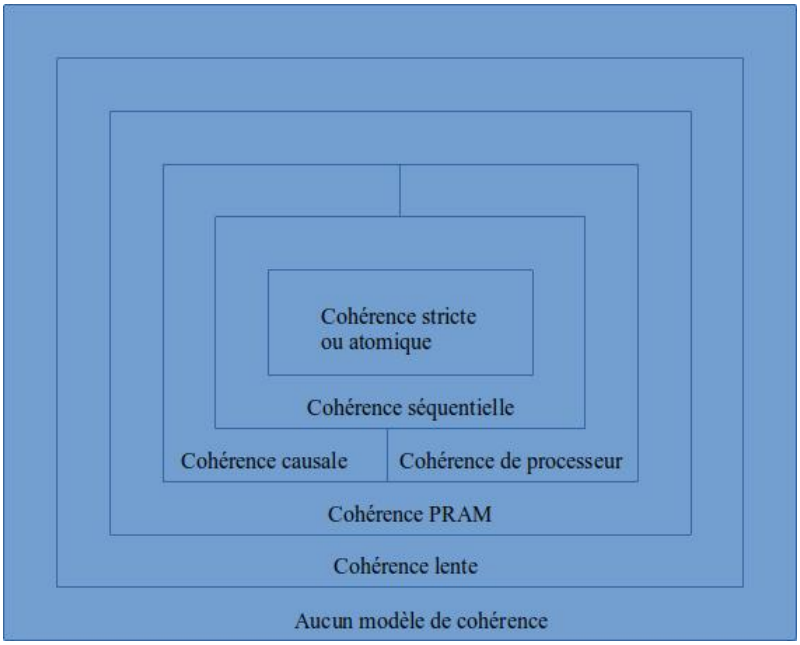

Figure 7.55 – Hiérarchie des modèles de cohérence

Type de cohérence	Description
Cohérence faible	Les données partagées sont cohérentes seulement après une opération de synchronisation.
Cohérence à la libération	Les données partagées sont cohérentes seulement à la sortie de la section critique.
Cohérence à l'entrée	Les données partagées sont cohérentes seulement à l'entrée d'une nouvelle section critique.

Table 7.2 – Modèles de cohérence avec synchronisation

Allègement	$W \to R$	$W \to W$	$R \to W/R$	RWE	ROE
SC				X	
IBM370	X				
TSO (Intel + SPARC)	X			X	
PC	X			X	X
PSO (SPARC)	X	X		X	
Alpha	X	X	X	X	
RMO (SPARC)	X	X	X	X	
PowerPC	X	X	X	X	X
ARMv7	X	X	X	X	X
RISC V	X	X	X	X	X
RC_{SC}	X	X	X	X	
RC_{PC}	X	X	X	X	X

Table 7.3 – Résumé des allègements

Le principal argument avancé pour ce type de système est celui qu'avec les adresses de 64 bits, l'espace d'adresses est tellement grand qu'il est possible d'en imaginer un seul pour tous les processus sur le réseau.

Concernant les espaces d'adresses de 64 bits

Que représente un espace d'adresses de 64 bits.
Selon Simpson [92], un espace de 64 bits emmagasinerait une vidéo dont la durée serait de 19 484 années, en supposant que chaque image occupe environ un mega-octet et que l'on enregistre 30 images par seconde. Donc, si nous avions lancé un enregistrement continue il y a 19 500 ans (pendant la dernière ère glacière), nous aurions finalement rempli la mémoire aujourd'hui.

Dans les années 90, lors de l'apparition des espaces d'adresses de 64 bits, certaines personnes les imaginaient quasi infinies. Ainsi, Chase et al [16] mentionnaient que si l'on chargeait des données en mémoire au rythme de un giga-octets par seconde, celle-ci serait remplie seulement au bout de 500 ans. Ils affirmaient qu'un tel espace était amplement suffisant pour la durée de vie (et plus) d'un seul ordinateur et suffisant pour une période de temps prolongée sur un réseau d'ordinateurs.

À l'époque, la vitesse des bus et des réseaux laissait croire que les adresses de 64 bits offraient effectivement un espace d'adresse presqu'impossible à remplir. Toutefois la technologie a évolué et les vitesses de transmission ont fait d'énormes avancées. Par exemple, en 2021, la mémoire DDR4 [29] autorisait un taux de transfert allant jusqu'à 25600 Méga-octets (ou 25,6 Giga-octets) par seconde et la mémoire DDR5 permettait un taux dépassant le 50 Giga-octets par seconde. Remplir un espace d'adresses n'est plus aussi «impossible» et les travaux sur les SASOS sont restés au point mort.

Toutefois, ces systèmes introduisaient des idées intéressantes. Les avantages présentés pour une telle architecture consistaient en :

- un changement de contexte plus rapide ;

En effet, cela ressemble à une architecture multi-fils mais étendue à tous.

- un partage plus facile des données et du code ;
  Un objet (une bibliothèque par exemple), se voit attribuer une adresse pour sa vie entière lors de sa création. Il n'est donc plus nécessaire de faire de l'édition des liens lors de la compilation des programmes. Le partage de données en serait facilité.
- une correspondance des fichiers en mémoire («memory mapped file») ;
  Il ne serait plus nécessaire d'écrire explicitement dans un fichier pour accomplir la persistance des données. La mémoire secondaire serait synchronisée avec la mémoire centrale. Ainsi toute donnée écrite à une certaine adresse serait automatiquement sauvegardée sur la mémoire secondaire.

Cependant, la protection des espaces d'adresses «privées» n'est plus possible avec la traduction d'adresses logiques. De nouveaux mécanismes de protection tout aussi efficaces ont été proposés.

Il existe de nombreux exemples de systèmes d'exploitation de types SASOS [110, 33, 44, 11, 159, 161]. La plupart sont des systèmes expérimentaux qui n'ont jamais été mis en production, mais des exceptions existent dont VxWorks [159], un système temps réel.

7.6 Gestion du temps [52, 103, 20, 106]

Une des caractéristiques des systèmes répartis est qu'il n'y a pas de concept de temps global ou unique. Chaque système possède sa propre horloge physique. Ces horloges évoluent à des rythmes différents et donc n'enregistrent pas la même heure.

Comme peut-on ordonnancer des événements dans le temps ou synchroniser des processus lorsque cet ordonnancement/synchronisation dépend de l'horloge (une horloge globale ou unique) ? Est-il possible d'établir une gestion de temps unique ou globale ?

7.6.1 Utilité d'une horloge globale

Une horloge globale remplit plusieurs rôles sur un réseau d'ordinateurs. Elles servent notamment :

- à la synchronisation de façon générale ;
- à la réservation de ressources (billets d'avion, ...) ;
- à la mise à jour dans les systèmes bancaire ;
- à la mise à jour dans les environnements de programmation ;
- au bon fonctionnement des jeux en réseau ;
- aux bases de données distribuées (journal, log, ...) ;
- au contrôle du trafic aérien ;
- à la simulation interactive.

7.6.2 Qu'est-ce que le concept de temps

Il est important de comprendre que le temps est un concept relatif et qu'il n'y a pas de «temps universel». Toutefois, pour bien fonctionner, nous avons besoin d'une notion de temps. Sur Terre, il est établi par une convention basée sur la rotation de la Terre en terme d'années, de jours et

d'heures. Ce temps est appelé Universal Coordinated Time (UTC). Par exemple, selon la période de l'année, Sherbrooke est à l'heure *UTC -4* (heure avancée) ou *UTC -5* (heure normale).

Les valeurs précises associées à ce temps sont aujourd'hui dérivées d'une horloge atomique.

7.6.3 Les ordinateurs et le temps

Les ordinateurs possèdent tous une horloge physique basée sur un cristal de quartz programmable (généralement pour générer des interruptions à toutes les 10ms) ayant une précision d'environ $1/10^6$ (1 seconde sur 11,6 jours).

Cette précision (ou ce manque de précision) signifie que les horloges de différents ordinateurs n'indiquent pas tous la même heure. Une solution viable pour augmenter la précision (et rendre possible l'utilisation des horloges pour la synchronisation) consiste à mettre à jour périodiquement ces horloges afin qu'elles indiquent la «bonne/même heure».

Il est possible d'acheter un ordinateur possédant un périphérique spécial capable de recevoir un signal UTC soit par ondes radio, par satellite ou par le réseau. La précision de l'heure sur le signal reçu dépend du médium. Ainsi, la précision par ondes radio est de 10 ms, 0,1 ms pour le service satellite GEOS (geostationnary operationnal environment satellite), 1 ms pour le service satellite GPS et de 1 millionième de seconde pour une horloge atomique.

Il s'avère toutefois impossible de munir tous les ordinateurs de ce type d'équipement. Aussi, seuls certains ordinateurs en sont dotés et agissent comme «serveur de temps». Malheureusement, la communication avec ces serveurs implique des délais.

Le problème demeure donc entier. Comment conserver un temps unique ou global même si les horloges dérivent et sont soumises à des délais dans les communications pour leur mise à jour ? Il existe deux solutions que nous explorerons dans cette section : les horloges logiques et la synchronisation des horloges physiques.

7.6.4 Les horloges logiques

Un système réparti est composé de processus qui collaborent vers l'atteinte d'un objectif commun. Comme il est difficile d'avoir une horloge physique globale, on fait appel à une autre méthode pour assurer l'ordonnancement : les horloges logiques.

Les horloges logiques sont des compteurs d'événements dont la mise à jour est basée sur la causalité et la relation arrivé-avant («happens-before») [125].

Définition : Horloge logique

Une horloge logique est un compteur d'événements dont la mise à jour est basée sur la causalité et la relation «s'est produit avant» (happens-before).

Soit deux événements a et b, la relation «arrivé-avant», notée $a \rightarrow b$, est une relation de cause à effet. Ainsi, sur un réseau d'ordinateurs, $a \rightarrow b$ si :

- a et b sont deux événements d'un même processus et a s'exécute avant b ;
- a est l'envoi d'un message par un processus P_1 et b est la réception de ce message par un processus P_2 ;
- il existe un événement c tel que $a \rightarrow c$ et $c \rightarrow b$ (transitivité).

La relation $a \rightarrow b$ crée un ordre partiel irréflexif sur les événements. Si a et b ne sont pas reliés par la relation $\rightarrow$, ils sont dits concurrents (a ne s'est pas produit avant b et b ne s'est pas produit avant a). En revanche, si $a \rightarrow b$ alors l'événement a peut affecter le résultat de l'événement b.

Soit la séquence d'événements présentée à la figure 7.56 où le temps s'écoule vers le bas (l'événement $p0$ s'est produit avant l'événement $p1$, etc). Selon la relation de précédence, il est possible de conclure que :

- $p1 \rightarrow q2$, $r0 \rightarrow q4$, $q3 \rightarrow r4$, $p1 \rightarrow q4$ ($p1 \rightarrow q2 \rightarrow q4$) ;
- $q0 \parallel p2$, $r0 \parallel q3$, $r0 \parallel p3$, $q3 \parallel p3$.

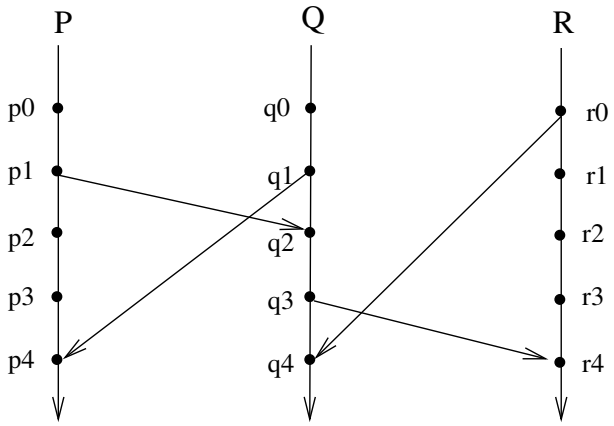

Figure 7.56 – Séquence d'événements dans le temps

Dans maintes situations, nul besoin d'ordonnancer les événements concurrents puisqu'ils sont indépendants. Toutefois dans certaines autres situations, il nécessaire d'obtenir un ordre total. Pour cela, il faut recourir aux les horloges logiques.

L'implantation la plus simple des horloges logiques a été proposée par Leslie Lamport[53, 156]. Elle respecte les règles suivantes :

- l'on associe une horloge logique (L_i) à chaque processus (P_i) et celle-ci est incrémentée à chaque événement local au processus ;

  Cette horloge ne mesure donc pas le temps mais constitue plutôt une sorte de compteur d'événements. Elle sert à associer une estampille (S) à chaque événement pour obtenir un ordre total des événements dans un processus ;

  Ainsi, si $a \rightarrow b$ alors $S(a) < S(b)$
- Pour étendre notre système à un ensemble de processus, on s'assure que chaque envoi de message possède une estampille inférieure à sa réception : $S(send(m)) < S(receive(m))$. Pour cela :
  - chaque envoi de message est estampillé, $S_i(send)$, d'après l'heure de l'horloge logique L_i du processus P_i qui fait l'envoi ;
  - lors de la réception de ce message par le processus P_j, l'horloge logique de P_j, L_j, est mise à jour de la façon suivante : $L_j = MAX(L_j + 1, S_i(send) + 1)$;

– La réception du message est estampillée avec cette horloge à jour (L_j).

La figure 7.57 présente un exemple de fonctionnement de cet algorithme.

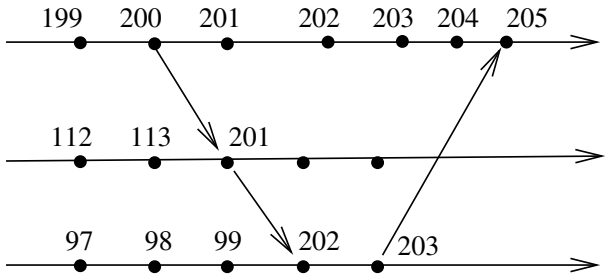

Figure 7.57 – Séquence d'événements avec leur estampille selon l'algorithme de Lamport

Les horloges logiques telles que définies par Lamport ne produisent pas un ordonnancement total des événements concurrents (qui ont la même valeur d'horloge) peuvent survenir. Dans cette situation, il suffit alors de s'entendre pour forcer un ordre sur les événements concurrents. Par exemple, si pour deux événements a et b, $S(a) = S(b)$, on emploie les identificateurs de processus pour briser l'égalité.

7.6.5 Horloges physiques

Les horloges logiques ne répondent pas à tous les besoins de synchronisation ou de coordination. Parfois, l'utilisation des horloges physiques est requise. Comme nous l'avons déjà mentionné, les horloges physiques ne progressent pas toutes à la même vitesse. En effet, les horloges physiques des ordinateurs basées sur un cristal de quartz ont tendance à dériver légèrement ce qui entraîne un décalage de temps entre elles. En 2008, une déviation de 40 microsecondes par seconde a même été rapportée [82].

Pour s'assurer que les différentes horloges physiques indiquent quasi toujours la même heure, il faut les synchroniser. Pour ce faire, il existe différents types d'algorithmes :

- les algorithmes centralisés ou décentralisés [25] ;
  Dans le cas des algorithmes centralisés, un seul ordinateur agit comme serveur/coordonnateur alors que pour les algorithmes décentralisés, plusieurs ordinateurs sont responsables de l'ajustement. Chaque machine diffuse aux autres la nouvelle heure. À partir de cette information, une moyenne est calculée et sert de nouvelle heure. Parfois, pour améliorer la moyenne, les valeurs extrêmes sont retirées du calcul.
- les algorithmes passifs ou actifs ;
  Avec les algorithmes dits passifs, les serveurs attendent les demandes d'heure alors qu'avec les algorithmes dits actifs, les serveurs interrogent les clients afin d'effectuer un ajustement de l'heure.
- les algorithmes qui se synchronisent avec un serveur de temps UTC (synchronisation externe) ou non (synchronisation interne) ;
- les algorithmes qui fonctionnent pour une masse de clients.

Peu importe le type d'algorithme utilisé, la synchronisation n'est pas toujours simple à réaliser. Il y a plusieurs facteurs à considérer :

- les délais de communication ;
  Lorsqu'une demande est envoyée sur le réseau pour un ajustement de l'heure, les délais de communication sont tels qu'à la réception de l'heure corrigée, celle-ci n'est déjà plus à jour. Cet état de fait exige alors un ajustement de l'heure reçue selon un certain délai de communication (calculé ou estimé). Certains environnements garantissent des bornes maximales et minimales sur les délais de communication. Cela aide à fixer l'ajustement.
- la transmission de l'heure ;
  Lorsqu'un ajustement est transmis au client, elle peut l'être sous deux formes. La première forme consiste à transmettre directement l'heure exacte. Le client n'a qu'à recopier l'heure ajustée selon les délais. La seconde forme consiste à transmettre l'écart entre l'heure du client et l'heure du serveur. Cette approche souffre moins des délais de transmission.
- l'ajustement de l'heure ;
  Lorsqu'une horloge physique décale, soit elle retarde, soit elle prend de l'avance. Certaines règles s'appliquent lors de cet ajustement.
  - Lorsque l'horloge prend de l'avance ;
    Normalement, il est interdit de retourner l'horloge en arrière dans le temps car cela est source d'ennuis pour les outils qui se basent sur l'heure pour prendre leurs décisions. En particulier, certains événements déclenchés par l'horloge pourraient l'être deux fois. Généralement on procède en «ralentissant» l'horloge temporairement jusqu'à l'atteinte de l'heure adéquate. Une façon de faire consiste à ajuster les incréments. On provoque, par exemple, les interruptions aux 11ms au lieu des 10ms normales et on incrémente l'horloge de 10ms à chaque interruption. Un autre choix serait de conserver les interruptions aux 10 ms mais d'incrémenter l'horloge de 9ms.
  - Lorsque l'horloge retarde ;
    Dans ce cas aussi un ajustement direct n'est pas recommandé. En effet, cela risque d'empêcher le déclenchement de certains événements. La solution consiste à accélérer l'horloge temporairement pour effectuer le rattrapage, exemple, en incrémentant l'horloge de 11 ms à chaque interruption, en supposant que celle-ci se produise toujours au 10ms.

Horloges physiques - synchronisation avec un serveur UTC

Ce type d'algorithme est basé sur la présence d'un serveur de temps spécialisé généralement équipé d'un récepteur UTC. L'algorithme de Christian [20] est de ce type. Les étapes suivantes sont requises pour synchroniser l'horloge d'un client.

- Le client interroge le serveur périodiquement afin d'ajuster l'heure (la période dépend de la précision désirée) ;
- Le serveur retourne l'heure ;
- Le client reçoit le message
  Le client a alors le choix d'utiliser l'heure directement ou de l'ajuster pour compenser les délais de communication en ajoutant un délai minimal connu ou la moitié du temps depuis l'envoi de la demande. Dans tous les cas, les actions à prendre sont :
  - Si la valeur du serveur > valeur de l'horloge locale
    Ajuster l'heure soit directement, soit en accélérant l'horloge temporairement pour effectuer le rattrapage.

– Si la valeur du serveur < valeur de l'horloge locale
  Ralentir l'horloge temporairement pour effectuer le rattrapage.

L'algorithme de Christian est un exemple de ce type d'algorithme. Il retourne directement la nouvelle heure aux clients.

Algorithme de Christian

L'algorithme de Christian est un algorithme passif. Il emploie un unique serveur connecté à une source UTC. Le serveur retourne au client la nouvelle heure. Le client ajuste l'heure en lui ajoutant un estimé du délai de communication.

ATS (Active Time Server)

L'algorithme ATS est un algorithme actif. Il utilise un unique serveur qui diffuse son heure à période fixe. Les clients se servent de cette heure pour ajuster leur horloge. Chaque client suppose un délai de communication fixe à ajouter à l'heure.

Horloges physiques - synchronisation sans serveur UTC

Il est possible de synchroniser les horloges d'un groupe d'ordinateurs sans recourir à un serveur. Pour y parvenir, les algorithmes dits centralisés utilisent l'un des ordinateurs en tant que coordonnateur. La synchronisation fonctionne de la façon suivante :

- Le coordonnateur demande le temps de chaque machine, calcule une moyenne et diffuse le résultat ;
- Les machines ajustent leur heure (comme avec un serveur UTC) ;
- Des ajustements manuels sont aussi envisageables.

Avec cette approche, les horloges des ordinateurs seront synchronisées entre elles, mais pas nécessairement précises par rapport au temps UTC. L'algorithme de Berkeley est un exemple de ce type d'algorithme.

Algorithme de Berkeley

L'algorithme de Berkeley est un algorithme actif centralisé basé sur un coordinateur (nommé) qui se charge de faire la synchronisation dans son groupe. Son mode de focntionnement est le suivant :

- Le coordinateur demande l'heure à tous les sites qu'il supervise ;
- Il calcule la moyenne en éliminant les valeurs considérées comme extrêmes ;
- Il retourne à chacun l'ajustement nécessaire (incrément ou décrément) ;
- Les clients ajustent leur heure.

Le mode de fonctionnement des algorithmes décentralisés est similaire, à l'exception qu'ici tous les ordinateurs se chargent de calculer la moyenne. Dans ce cas, tous les sites diffusent leur heure à tous les autres sites. Chacun des sites se charge ensuite de calculer leur «nouvelle heure» et de mettre à jour leur horloge.

Global Averaging Algorithm (GAA)

Selon l'algorithme GAA, chaque noeud diffuse à période fixe son heure locale sous la forme d'un message spécial de synchronisation (resync). Après la diffusion, chaque noeud attend pendant un certain temps, temps durant lequel il collecte les heures provenant des autres sites. À la fin de la période d'attente, il estime le décalage de son horloge par rapport à celles des autres sites et effectue la correction.

Local Averaging Algorithm (LAA)

Cet algorithme est similaire à GAA. Il corrige toutefois certains problèmes de mise à l'échelle de GAA grâce à un algorithme de diffusion par commérage. Ainsi, chaque site possède un certain nombre de voisins. À période fixe, chaque noeud échange son heure locale avec ses voisins et ajuste son horloge selon la moyenne de toutes les heures recueillies, y compris la sienne.

Horloges physiques - grande échelle

Un seul serveur ne suffit généralement pas à servir un grand nombre de clients. Pour des raisons de fiabilité et de performance, le recours à une hiérarchie de serveurs est souvent préférable. Le protocole NTP (Network Time Protocol) [157, 94, 85, 25] a été conçu à cette fin. Il est aussi en mesure de synchroniser les horloges à quelques dizaines de millisecondes près.

Pour parvenir à ce résultat, NTP fait appel à une hiérarchie de serveurs formant une structure en arbre et plusieurs modes de diffusions de l'heure. Les deux modes les plus fréquemment employés sont le mode passif de style client/serveur pour la diffusion verticale dans la hiérarchie et le mode symétrique actif/passif pour la diffusion de l'heure entre ordinateurs d'un même niveau.

La hiérarchie du protocole NTP comprend au moins trois niveaux (appelés «*Stratum*») :

- Niveau 0 («*Stratum 0*») : les horloges physiques ;

  Ce sont des horloges de haute précision tels des horloges atomiques, GPS ou ondes radio. Elles génèrent des signaux permettant d'obtenir une heure avec un très haut degré de précision. Elles sont connectées directement à un ordinateur du niveau 1. Ce niveau ne contient aucun serveur, seulement des périphériques (UTC).
- Niveau 1 («*Stratum 1*») : les serveurs de temps primaires (la racine) ;

  À ce niveau, on retrouve les serveurs primaires connectés directement et synchronisés aux récepteurs UTC (aux horloges physiques du niveau 0). Ceux-ci propagent ensuite l'heure au niveau supérieur, i.e. les serveurs secondaires ou les clients (en mode client/serveur).

Les serveurs de ce niveau ont la capacité de s'associer à d'autres serveurs du même niveau pour des raisons de fiabilité et de validation (en mode symétrique passifs/actifs).

- Niveau 2 («*Stratum 2*») : les serveurs secondaires ;
  Au second niveau de chaque sous-arbre, un groupe de serveurs interagissent avec les serveurs du niveau 1. Un serveur de niveau 2 peut interagir avec plusieurs serveurs de niveau 1 afin de sélectionner l'heure la plus précise. Ils ont aussi la possibilité de s'associer avec d'autres serveurs du niveau 2 en mode symétrique actif/passif.
  Chaque second niveau diffuse ensuite l'heure au troisième niveau
- Niveau 3 («*Stratum 3*») : les clients (généralement) ;
  Les ordinateurs de ce niveau se synchronisent soit avec un ou plusieurs serveurs du niveau 2, soit entre eux avec le mode symétrique.
  S'il y a plus de 3 niveaux, ils agiront comme serveurs pour la couche suivante.
- Les couches suivantes.
  Cette hiérarchie s'étend potentiellement jusqu'à 15 niveaux. Toutefois la précision de l'heure diminue généralement au fur et à mesure que l'on s'éloigne de la racine.

La figure 7.58 illustre la hiérarchie propre au protocole NTP.

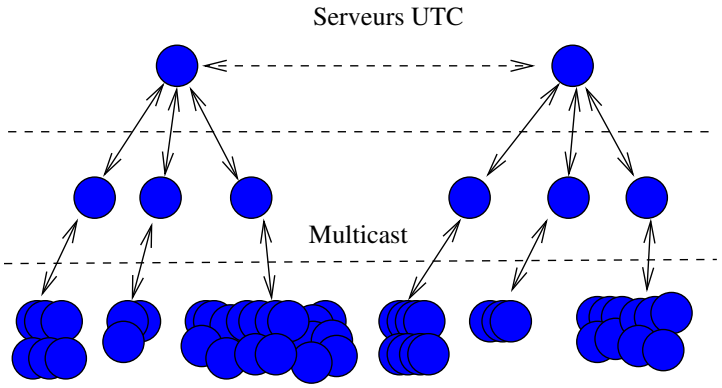

Figure 7.58 – Hiérarchie de serveurs du protocole NTP

Une version simplifiée de NTP, SNTP (Simple Network Time Protocol) sert également dans des environnements n'ayant pas la capacité de supporter la version complète de NTP. Le protocole SNTP est NTP moins quelques composants lui permettant ainsi de consommer moins de ressources (mémoire, UCT, ...). Il offre cependant moins de précision que NTP.

Synchronisation des horloges dans un environnement sans fil

À venir !

7.7 Synchronisation

Nous avons déjà abordé le concept de synchronisation dans les systèmes centralisés, en particulier l'exclusion mutuelle. Mais comment l'implanter dans un environnement réparti ?

Modélisons un système réparti comme un ensemble de processus (un par machine), numérotés de 1 à n, ne partageant aucune mémoire. Pour simplifier, admettons que chaque processus représente un ordinateur du réseau.

Il y a plusieurs approches pour implanter l'exclusion mutuelle sur un système réparti.

7.7.1 Approche centralisée

Selon l'approche centralisé, un processus sert de coordonnateur. Un processus désirant entrer en exclusion mutuelle envoie un message de type *demande* au coordonnateur. Lorsque le processus demandeur reçoit la *réponse*, il entre en section critique. Lorsqu'il en sort, il envoie un message de type *libération* au coordonnateur.

Lorsque le coordonnateur reçoit un message de type *demande*, il vérifie si un processus occupe déjà la section critique. Si celle-ci est disponible, il envoie une *réponse*. Si la section critique n'est pas disponible, la demande est mise en attente (file d'attente). Lorsque le coordonnateur reçoit un message de type *libération*, il retire une demande de la file d'attente et envoie une *réponse*.

Cet algorithme assure l'exclusion mutuelle et l'équité (dépendamment de la façon dont la file d'attente est gérée). Cette approche nécessite trois messages par entrée en section critique.

Qu'arrive-t-il si le coordonnateur tombe en panne ? Il doit y avoir élection et le nouveau processus coordonnateur interroge les autres processus pour reconstruire la file d'attente.

7.7.2 Approche distribuée

L'algorithme précédant présente plusieurs inconvénients dus au recours à un coordonnateur unique (performance et fiabilité). Une approche distribuée, sans coordonnateur, règle certains de ces problèmes. L'un des algorithmes proposés [81] nécessite la présence d'horloges logiques et d'un système de communication de groupe (multicast).

Lorsqu'un processus P_i veut entrer en section critique, il génère une estampille E et envoie un message $demande(P_i, E)$ à tous les processus du système s'incluant lui-même. Lorsqu'une *demande* est reçue, un processus peut répondre immédiatement ou attendre selon certains critères :

- Si le processus est en section critique, il met la demande en attente ;
- Si le processus ne souhaite pas accéder à la section critique, il envoie une réponse immédiatement ;
- Si le processus désire entrer en section critique mais qu'il n'a pas reçu toutes les réponses, il compare l'estampille de sa demande avec celle de la demande reçue :
  - Si son estampille est plus grande, il envoie une réponse ;
  - Si son estampille est plus petite, il met la demande en attente.

Un processus qui a reçu des réponses de tous les processus a la permission d'entrer en section critique. Il placera alors toutes les demandes qu'il reçoit en attente.

Illustrons le fonctionnement de cet algorithme à l'aide d'exemples. Soit quatre processus : P_1, P_2, P_3 et P_4. Supposons que P_1 veuille accéder à la section critique. Il génère une demande «D» avec une estampille «$T = 4$» (figure 7.59 a). P_1 envoie sa demande $D_1(4)$ aux autres processus

(figure 7.59 b). Les processus P_2, P_3, et P_4, ne désirant pas obtenir la section critique, envoient une réponse positive, soit $R_i(ok)$, à P_1 (figure 7.59 c). Ce dernier entre donc en section critique (figure 7.59 d). Finalement, le processus P_1 termine sa section critique et poursuit sa tâche.

P1	P2	P3	P4
D (T=4)			

a P_1 génère une demande d'entrée en SC.

P1	P2	P3	P4
D (T=4)	$D_1(4)$ →ok	$D_1(4)$ →ok	$D_1(4)$ →ok

b P_1 envoie sa demande aux autres processus.

P1	P2	P3	P4
$D(T=4)$			
	$D_1(4)$ →ok	$D_1(4)$ →ok	D1(4)→ok
$R_2(ok)$			
$R_3(ok)$			
$R_4(ok)$			

c P_2, P_3 et P_4 envoient leur réponse.

P1	P2	P3	P4
$D(T=4)$			
	$D_1(4)$ →ok	$D_1(4)$ →ok	$D_1(4)$ →ok
$R_2(ok)$			
$R_3(ok)$			
$R_4(ok)$			
SC			

d P_1 entre en section critique.

P1	P2	P3	P4
$D(T=4)$			
	$D_1(4)$ →ok	$D_1(4)$ →ok	$D_1(4)$ →ok
$R_2(ok)$			
$R_3(ok)$			
$R_4(ok)$			
SC			
...			

e P_1 sort de la section critique et poursuit son exécution.

Figure 7.59 – Exemple 1 : exclusion mutuelle distribuée.

Cette fois, illustrons la situation dans laquelle plusieurs processus souhaitent avoir accès à la section critique en même temps. Soit quatre processus : P_1, P_2, P_3 et P_4. Supposons que P_1 et P_3 désirent entrer en section critique. P_1 génère une demande «D_1» avec une estampille «$T = 46$» et P_3 génère une demande «D_3» avec une estampille «$T = 9$» (figure 7.60 a). P_1 et P_3 envoient ensuite leur demande aux autres processus (figure 7.60 b). P_2 et P_4 répondent positivement aux demandes car ils ne sont pas intéressés par la section critique. Quant au processus P_3, il répond positivement car son estampille est plus grande ($T_1 < T_3$). Le processus P_1 met la demande de P_3 en attente car l'estampille de P_3 est plus grande ($T_1 < T_3$) (figure 7.60 c). P_1 entre donc en premier en section critique (il a reçu toutes les réponses) et P_3 se met en attente (il n'a pas reçu de réponse de P_1) (figure 7.60 d). Finalement, P_1 sort de la section critique, répond à P_3 et poursuit son exécution (figure 7.60 e). P_3 reçoit la réponse de P1 et entre en section critique (figure 7.60 f). Enfin, P_3 sort de sa section critique et poursuit son exécution(figure 7.60 g).

Cet algorithme assure l'exclusion mutuelle sans interblocage et est équitable.

Toutefois, il nécessite un nombre élevé de messages pour obtenir l'autorisation d'entrer en section critique, soit $2 \times (n - 1)$. Si un système de diffusion de groupe existe, le nombre de messages est diminué par deux. Certaines autres optimisations sont possibles [20]. De plus, il faut connaître les

P1	P2	P3	P4
$D_1(4)$		$D_3(9)$	

a P_1 et P_3 génèrent des demandes d'entrée en SC.

P1	P2	P3	P4
$D_1(4)$		$D_3(9)$	
	$D_1(4)$ →ok	$D_1(4)$ →ok	$D_1(4)$ →ok
$D_3(9)$ →AT	$D_3(9)$ →ok		$D_3(9)$ →ok

b P_1 et P_3 envoient leur demande.

P1	P2	P3	P4
$D_1(4)$		$D_3(9)$	
	$D_1(4)$ →ok	$D_1(4)$ →ok	$D_1(4)$ →ok
$D_3(9)$ →AT	$D_3(9)$ →ok		$D_3(9)$ →ok
$P_2(ok)$		$P_2(ok)$	
$P_3(ok)$		$P_4(ok)$	
$P_4(ok)$			

c P_1, P_2, P_3 et P_4 envoient leur réponse.

P1	P2	P3	P4
$D_1(4)$		$D_3(9)$	
	$D_1(4)$ →ok	$D_1(4)$ →ok	$D_1(4)$ →ok
$D_3(9)$ →AT	$D_3(9)$ →ok		$D_3(9)$ →ok
$P_2(ok)$		$P_2(ok)$	
$P_3(ok)$		$P_4(ok)$	
$P_4(ok)$			
SC			

d P_1 entre en section critique.

P1	P2	P3	P4
$D_1(4)$		$D_3(9)$	
	$D_1(4)$ →ok	$D_1(4)$ →ok	$D_1(4)$ →ok
$D_3(9)$ →AT	$D_3(9)$ →ok		$D_3(9)$ →ok
$P_2(ok)$		$P_2(ok)$	
$P_3(ok)$		$P_4(ok)$	
$P_4(ok)$			
SC			
...		P1 (ok)	
		SC	

e P_1 sort de sa section critique et répond à P_3.

P1	P2	P3	P4
$D_1(4)$		$D_3(9)$	
	$D_1(4)$ →ok	$D_1(4)$ →ok	$D_1(4)$ →ok
$D_3(9)$ →AT	$D_3(9)$ →ok		$D_3(9)$ →ok
$P_2(ok)$		$P_2(ok)$	
$P_3(ok)$		$P_4(ok)$	
$P_4(ok)$			
SC			
...		P1 (ok)	
		SC	
		...	

f P_3 entre en section critique.

g P_3 poursuit son exécution.

Figure 7.60 – Exemple 2 : exclusion mutuelle distribuée.

processus impliqués. L'ajout ou le retrait d'un processus est complexe.

Cet algorithme est-il résistant aux pannes ? Qu'arrive-t-il si un processus tombe en panne ?

7.7.3 Passage de jeton (Token ring)

Pour que cet algorithme fonctionne, tous les processus doivent d'abord être organisés en un anneau logique. Chaque processus reçoit un identificateur qui représente sa position dans l'anneau. Ainsi, le processus i communique seulement avec ses voisins, soient les processus $i-1$ et $i+1$.

Pour assurer l'exclusion mutuelle, un message spécial, appelé le jeton, circule sur l'anneau. La possession du jeton permet l'accès à la section critique. Lorsqu'un processus ne désire pas y entrer (ou en sort), il envoie tout simplement le jeton à son voisin dans l'anneau. L'exclusion mutuelle est garantie par la présence d'un seul jeton en circulation. Cet algorithme est équitable si l'anneau est unidirectionnel, i.e. le jeton circule toujours dans le même sens.

Le nombre de messages échangés pour obtenir un droit d'accès à la section critique varie entre 0 et n (la taille de l'anneau).

La tolérance aux pannes est cependant plus complexe à gérer.

- Perte du jeton ;

  On doit d'abord élire un processus responsable de la régénération du jeton. Cela fait, le jeton sera régénéré. Après coup, plusieurs jetons seront possiblement en circulation (si le jeton perdu est retrouvé !), alors l'un des jetons devra être éliminé.

- Panne d'un processus.

  Quand un processus tombe en panne, l'anneau doit être reconstruit pour «contourner» ce processus. Lorsqu'un processus en panne redevient actif, il doit être réintégrer dans l'anneau.

7.7.4 Algorithme de votation

Maekawa[61, 20] a profité du fait qu'il ne soit pas nécessaire qu'un processus obtienne l'approbation de tous les autres processus pour entrer en section critique. Il suffit qu'un processus obtienne l'accord d'un sous-ensemble des pairs, à condition que tous les sous-ensembles utilisés par les processus se chevauchent. Cela ressemble à un système de votation. Un candidat pour la section critique doit cumuler suffisamment de vote pour y accéder. Comme un processus vote pour un seul candidat et que les ensembles se chevauchent, alors un processus unique obtiendra éventuellement suffisamment de votes pour entrer en section critique.

7.7.5 Transactions atomiques

Les techniques de synchronisation que nous avons abordées jusqu'à maintenant afin d'assurer l'exclusion mutuelle, sont essentiellement des outils de bas niveau. La personne en charge du développement doit s'occuper elle-même des détails de l'exclusion mutuelle, de la prévention des interblocages et de la reprise après une panne. Les transactions atomiques sont en fait une abstraction de haut niveau qui cache les problèmes techniques associés à l'exclusion mutuelle, aux interblocages et à la reprise.

Ce concept a déjà été présenté brièvement dans le cadre de la gestion des fichiers répartis ainsi que dans les cours de base de données.

Une transaction consiste en une collection d'opérations qui exécutent une fonction logique et unique, comme la mise à jour d'un compte en banque. C'est donc un programme qui accède et

met à jour possiblement plusieurs éléments de donnés pouvant résider sur différents disques, voire sur différents sites. Une transaction se définit donc généralement comme une simple séquence de lectures et d'écritures qui se termine par un engagement ou une annulation.

Définition : Transaction

Une transaction est une abstraction de haut niveau qui effectue, en simulant une seule opération logique, une série d'accès et de mises à jour à des éléments de données.

Une transaction est atomique si toutes les opérations qui lui sont associées sont exécutées entièrement ou si aucune d'entre elles ne l'est. Les solutions au problème de synchronisation que nous avons présentées jusqu'à maintenant ne conviennent pas car elles ne tiennent pas compte de la possibilité de pannes à l'intérieur d'une «section critique» d'un ou de plusieurs processus impliqués. Comment peut-on donc assurer que l'exécution des opérations en section critique soit atomique quand certains processus risquent de tomber en panne ? De plus, implanter une transaction atomique sur un système réparti se révèle complexe car plusieurs sites ont la possibilité de participer à la transaction (mise à jour de copies multiples). La panne d'un ou plusieurs sites ne doit en aucun cas nuire à l'atomicité de la transaction.

Définition : Transaction atomique

Une transaction est atomique si toutes les opérations qui lui sont associées sont exécutées entièrement ou si aucune d'elles n'est exécutée.

Pour implanter les transactions atomiques, on a recourt aux coordonnateurs de transactions. Chaque site possède un tel coordonnateur responsable des transactions démarrées localement. Il doit :

- démarrer la transaction locale ;
- diviser la transaction en sous-transactions ;
- distribuer les sous-transactions vers les sites appropriés ;
- coordonner la fin de la transaction (engagement) ;

La principale difficulté dans ces tâches consiste justement à coordonner la fin de la transaction. Pour cela, chaque site (ou coordonnateur) utilise un journal écrit sur une mémoire stable. Rappelons qu'une information qui réside en mémoire stable n'est jamais, par définition, ni perdue ni corrompue et ce même en présence de pannes matérielles ou logicielles. Pour plus d'informations sur les mémoires stables, nous vous référons à l'annexe A. La mémoire stable est un élément essentiel permettant de garantir l'atomicité d'une transaction.

Comme tous les sites doivent s'entendre sur le résultat final de la transaction (engagement ou annulation), l'un des coordonnateurs est responsable d'exécuter un protocole d'engagement afin de coordonner la fin d'une transaction atomique. Bien qu'il existe plusieurs protocoles d'engagement, le protocole d'engagement à deux phases est le plus courant.

Protocole d'engagement à deux phases

Soit T une transaction initiée à un site S_i par un coordonnateur C_i. Lorsque T complète son exécution, C_i devient responsable de l'exécution du protocole d'engagement.

Lors de la première phase (**Phase 1**) de ce protocole :

- C_i enregistre $<prepare\ T>$ dans son journal en mémoire stable ;
- C_i envoie $<prepare\ T>$ à tous les sites ;
- un site qui reçoit ce message décide de s'engager ou non :
  - s'il ne s'engage pas, il enregistre $<non\ T>$ dans son journal (mémoire stable) puis envoie $<non\ T>$ à C_i ;
  - s'il s'engage, il enregistre $<ok\ T>$ et tous les éléments de T dans son journal, puis envoie $<ok\ T>$ à C_i ;
    Pour toutes les opérations d'écriture, il faut enregistrer au préalable dans le journal l'identificateur de l'opération, l'identificateur de la donnée à modifier, l'ancienne valeur et la nouvelle valeur. Une entrée de ce type dans le journal prend donc la forme :
    « *Transaction T veut modifier la donnée D de la valeur V_1 vers la valeur V_2* »

L'envoi des réponses par tous les sites impliqués termine la phase 1. La phase 2 débute lorsque C_i a reçu toutes les réponses (ou un dépassement de délai - timout).

Lors de la **phase 2** :

- C_i cumule les réponses et détermine si on engage ou annule la transaction :
  - Si engagement, il enregistre $<engagement\ T>$ dans son journal et envoie $<engagement\ T>$ à tous les sites ;
    L'engagement est initié seulement si toutes les réponses des sites sont $<ok\ T>$.
  - Si annulation, il enregistre $<annulation\ T>$ dans son journal et envoie $<annulation\ T>$ à tous les sites.
    L'annulation est initiée dès qu'un seul site refuse de s'engager ($<non\ T>$).
- Lorsqu'un site reçoit ce message, il l'enregistre dans le journal puis exécute l'action demandée.

Selon ce protocole, un site peut annuler une transaction T en tout temps avant son engagement. Toutefois une fois l'engagement expédié, il est tenu de compléter la transaction même en présence de pannes. C'est pour cette raison que l'engagement est toujours précédé de l'enregistrement de toutes les informations nécessaires sur une mémoire stable afin d'exécuter ou de ré-exécuter la transaction T.

De même, selon ce protocole, dès qu'un seul site refuse de s'engager, la transaction est entièrement annulée.

Le fait d'enregistrer toutes les informations dans le journal permet de rendre la transaction atomique. En effet, cet enregistrement est suffisant pour retourner en arrière si cela est nécessaire (point de contrôle ou *checkpoint*) et/ou pour reprendre et poursuivre un traitement qui aurait été interrompu. Ces informations sont particulièrement utiles en cas de panne. Ainsi, si une panne se produit au niveau :

- d'un site ;
  Lors de la reprise, le site doit examiner son journal et effectuer les actions suivantes :
  - Si le journal contient $< engagement\ T >$, on tente de compléter la transaction. Pour cela on ré-exécute toutes les actions enregistrées dans le journal ($redo(T)$).
  - Si le journal contient $< annulation\ T >$, on remet les données dans leur état antérieur ($undo(T)$). Pour cela, il faut, dans certains cas, récupérer les anciennes valeurs emmagasinées dans le journal.
  - Si le journal contient $< ok\ T > \rightarrow$, le site doit consulter le coordonnateur C_i pour connaître la décision. Selon l'état retourné par C_i,
    - $< engagement\ T >$, on complète la transaction ($redo(T)$) ou
    - $< annulation\ T >$, on l'annule ($undo(T)$).

Si C_i ne répond pas (il est en panne), il faut contacter les autres sites pour connaître la décision. Si aucune information n'est disponible, le site doit attendre et interroger périodiquement le coordonnateur C_i.
  - Si le journal ne contient rien, on annule la transaction ($undo(T)$).
- du coordonnateur ;
  Lorsque le coordonnateur C_i tombe en panne, ce sont les sites participants qui doivent décider. Ainsi :
  - Si au moins un site contient $< engagement\ T >$, alors la transaction est engagée et tous les sites doivent la compléter.
  - Si au moins un site contient $< annulation\ T >$, alors la transaction est annulée.
  - Si quelques sites ne contiennent pas $< ok\ T >$, alors la transaction T est annulée ;
  - Si tous les sites contiennent $< ok\ T >$, il est impossible de connaître la décision de C_i. Il faut alors attendre le retour du coordonnateur C_i. Le problème provient du fait que les ressources sont verrouillées pendant toute cette période.
- du réseau.
  Il est possible qu'un ou plusieurs liens du réseau tombent en panne. La panne d'un seul lien apparaît généralement comme une panne d'un site ou du coordonnateur. Les actions à entreprendre sont donc celles prévues pour ces cas.
  La panne de plusieurs liens entraînent le partitionnement du réseau et la création de sous-réseaux . Cela apparaît, encore une fois, telle une panne d'un site ou du coordonnateur

7.7.6 Transactions concurrentes

Lorsque plusieurs transactions s'exécutent simultanément, un conflit peut survenir lors de l'accès aux données. Une forme de synchronisation est nécessaire pour éviter la corruption des données. Celle-ci n'a pas besoin d'être aussi stricte que l'exclusion mutuelle. Il est suffisant d'assurer la «sérialisabilité» des transactions. Cela signifie que lorsque deux transactions s'exécutent simultanément, le résultat de leur exécution doit être le même que si elles s'étaient exécutées séquentiellement. Une solution consiste à recourir à :

- des verrous (répartis, centralisés, protocole de verrouillage, ...)
  L'utilisation de verrous est l'approche la plus ancienne et la plus courante. La solution la plus simple consiste à verrouiller chaque enregistrement avant de l'accéder, i.e. au début de la transaction et de le relâcher seulement à la fin de la transaction. Cela se fait soit par l'intermédiaire d'un gestionnaire centralisé de verrous, soit par des gestionnaires répartis sur chacun des sites.
  Cette approche étant restrictive, des optimisations sont envisageables, comme faire une distinction entre les verrous de lecture et ceux d'écriture. Il est possible de raffiner encore davantage en ne verrouillant, par exemple, qu'un sous-ensemble des fichiers plutôt que des fichiers entiers. Ces améliorations augmentent le parallélisme mais au prix d'un plus grand nombre de verrous et d'une complexité accrue pour leur gestion.
  Une autre optimisation consiste à ne pas maintenir un verrou pendant toute la transaction. Pour y parvenir il faut suivre un protocole de verrouillage strict. Le plus populaire est le protocole de verrouillage à deux phases avec lequel tous les verrous sont obtenus pendant la première phase, phase durant laquelle aucun verrou ne peut être libéré. La seconde phase débute lorsque la transaction commence à libérer des verrous. Il est alors permis de libérer des verrous mais interdit d'en acquérir de nouveaux. Ce protocole assure ainsi la sérialisabilité. Il

ne garantit toutefois pas l'absence d'interblocage. De plus, sous ce protocole certains conflits difficiles voire impossibles à résoudre se manifestent.
Lorsque l'on considère les transactions atomiques, il est nécessaire de conserver les verrous tant qu'une transaction n'est pas engagée ou annulée.

- une approche optimisme ;
  Un autre façon de gérer la concurrence est de procéder par ce qu'on appelle «le contrôle optimiste». L'idée de base ici est de ne pas intervenir immédiatement, i.e. laisser les transactions s'occuper de leurs modifications et, éventuellement à la toute fin, si nécessaire, de résoudre les conflits. Comme en pratique ceux-ci sont relativement rares, cela n'exige aucune action dans la majorité des cas. Lorsqu'un conflit se produit, on le règle en annulant la transaction responsable du conflit. Celle-ci se détecte en vérifiant, au moment de l'engagement, si l'un des enregistrements visés a été modifié depuis le début de la transaction. Si c'est le cas, la transaction est annulée.
- des estampilles ;
  Une autre approche consiste à assigner à chaque transaction T_i une estampille $TS(T_I)$ lorsqu'elle débute son exécution. On s'assure par la suite que la transaction T_i s'exécute avant T_j si $TS(T_i) < TS(T_J)$. La priorité serait donc octroyée à T_i lors d'un conflit (la transaction T_i est plus ancienne puisque son estampille est plus petite).
  Pour y parvenir, on conserve une trace des plus récentes transactions d'écriture et une autre pour celles de lecture sur chaque donnée D, nommées respectivement $TS_E(D)$ et $TS_L(D)$.
  - Validation lors d'une lecture
    Si la transaction T_i s'apprête à lire une donnée D modifiée par une transaction plus récente ($TS(T_i) < TS_E(D)$), alors T_i est annulée (ses modifications renversées) afin qu'elle reprenne son exécution avec une nouvelle estampille. Si la transaction T_i s'apprête à lire une donnée D modifiée par une transaction plus ancienne ($TS(T_i) > TS_E(D)$), alors T_i effectue sa lecture et l'âge de T_i est comparé à celle de la dernière lecture sur D. Si la transaction T_i est plus récente, alors la date du dernier accès à D est mise à jour avec son estampille.
  - Validation lors d'une écriture
    Si la transaction T_i s'apprête à modifier une donnée D lue par une transaction plus récente ($TS(T_i) < TS_L(D)$), alors T_i est annulée (ses modifications renversées) afin qu'elle reprenne son exécution avec une nouvelle estampille. Si la transaction T_i s'apprête à mettre à jour une donnée D déjà modifiée par une transaction plus récente ($TS(T_i) < TS_E(D)$), alors T_i tente d'écrire une valeur périmée. T_i est alors annulée (ses modifications renversées) afin qu'elle reprenne son exécution dotée d'une nouvelle estampille. Dans tous les autres cas, T_i effectue sa mise à jour et la date de la dernière modification à D est ajustée avec son estampille.

  Ce protocole assure que les opérations conflictuelles soient exécutées dans le même ordre que celui des estampilles. La sérialisabilité est donc effective. De plus, aucun interblocage n'est en mesure de se produire vu l'absence d'attente (la transaction procède ou recommence).
  Il est toutefois possible qu'une transaction souffre de famine (la transaction serait continuellement reprise due à des conflits d'estampilles).

7.7.7 Élections[20, 52, 103]

Dans un système réparti, il arrive parfois que l'accomplissement d'une tâche exige la présence d'un processus coordonnateur. Cela se produit, par exemple, lors de la reprise et de la reconfiguration d'un réseau suite à une panne d'un processus. Dans cette situation, une phase dite d'élection est nécessaire pour désigner le processus coordonnateur. Avant l'élection, aucun des processus ne sait lequel d'entre eux agira comme coordonnateur, ni n'est capable de communiquer avec ce dernier. Après l'élection, tous connaîtront l'identité de l'unique processus élu.

Plusieurs algorithmes permettent d'élire un coordonnateur. Le problème se résume par le fait que chacun des processus tente ou non de se faire nominer pour agir en tant que coordonnateur, sujet à la contrainte qu'un seul processus accédera à ce poste. Un algorithme résolvant ce problème est acceptable si toutes les conditions suivantes sont satisfaites :

- tous les processus sont soit dans l'état «élu», soit «non-élu» mais une fois qu'un processus est dans l'état élu, il le demeure.
- il termine, i.e. qu'après un temps fini, le coordonnateur est sélectionné ;
- pour une exécution particulière, un seul processus parvient à l'état «élu».
- après l'élection, tous les processus connaissent l'identité du nouveau coordonnateur.

Évidemment les algorithmes d'élection varient selon la topologie du réseau sous-jacent.

Une technique simple pour élire un coordonnateur est de choisir le processus ayant le plus grand «indice». Le choix des valeurs qui serviront d'indices importe peu tant qu'elles sont uniques et totalement ordonnées. Par exemple, il est possible de choisir la charge de travail comme indice et d'élire le processus dotée de la charge la plus faible. Si deux processus possédaient la même charge de travail, les identificateurs de processus pourraient alors servir à les départager.

Voici deux algorithmes :

- Un algorithme sur un anneau logique (ring-based election algorithm)
  Selon cette topologie, chaque processus possède un canal de communication avec son successeur dans l'anneau. Tous les messages sont transmis dans une seule direction sur l'anneau (unidirectionnel). Selon cet algorithme, le processus ayant le plus grand identificateur est élu. Un processus initie l'élection en plaçant son identificateur dans un message de type élection et le fait parvenir à son successeur. Quand un processus reçoit un message d'élection, il compare son identificateur avec celui contenu dans le message. Si son indice est plus petit, il transmet le message directement. Si son indice est plus grand, il remplace l'indice reçu par le sien et renvoie le message. Si l'indice est le sien, il devient le coordonnateur et envoie alors un message de fin d'élection contenant son identité à ses voisins.
- l'algorithme d'intimidation (Bully algorithm)
  Cet algorithme permet de supporter les pannes de processus pendant une élection (détectables avec des horloges de garde). Il suppose aussi que chacun des processus connaît les processus ayant un indice supérieur au sien et qu'il peut communiquer avec eux.
  Cet algorithme utilise trois types de messages. Un message d'élection est requis pour initier une élection, une réponse est retournée à ce dernier message et, pour terminer, un troisième type de message (une annonce) sert à faire connaître l'identité du nouveau coordinateur.
  Une élection débute lorsqu'un processus détecte la panne du coordonnateur. Si le processus possède le plus grand identificateur, il se déclare nouveau coordonnateur et transmet aussitôt un message de type «annonce» à tous les processus. Si le processus ne possède pas le plus grand identificateur, il débute une élection en faisant parvenir un message de type «élection» aux processus ayant un identificateur plus élevé puis attend une réponse. S'il n'en reçoit pas

après un délai fixé, il se déclare coordonnateur et le communique par un message de type «annonce» à tous les processus. Si une réponse lui parvient, il patientera jusqu'à la réception d'un message de type «annonce», sinon sans l'arrivée de ce dernier (après un certain délai), il débute une nouvelle élection.
Si un processus reçoit un message annonce, il reconnaît le nouveau coordonnateur (change sa configuration). S'il reçoit un message d'élection, il retourne une réponse et débute une autre élection.
Lorsqu'un processus «revient» après un panne, il démarre une nouvelle élection. S'il possède le plus grand identificateur, il se déclare coordonnateur même si l'existant est encore fonctionnel (il le remplace).

7.7.8 Consensus et accord (agreement) [20, 52, 151]

Dans un système parallèle ou distribué, les processus doivent fréquemment s'entendre sur une valeur commune. Le problème dit du consensus, peut s'avérer très complexe à résoudre selon les hypothèses fixées, en particulier celles considérant les pannes propres ou impropres. Une panne propre, du réseau ou d'un site, désigne le cas où celui-ci cesse de fonctionner. Les messages sont alors perdus ou tout simplement, ne sont plus transmis. Une panne impropre, encore une fois d'un site ou du réseau, définit le cas où, plutôt que de cesser complètement de transmettre, le site (ou le réseau) transmet des valeurs erronées dans les messages. Cette dernière situation est aussi qualifiée de «**panne byzantine**». Ce type de panne a été introduit sous l'allégorie «problème des généraux Byzantins» [55] et le champs d'étude sur la tolérance aux pannes byzantines (BFT) tente d'apporter des solutions à ce problème.

Les pannes byzantines[20, 52, 142, 149, 101, 65, 22, 23]

Définition : Panne byzantine[142, 149]

En informatique, on appelle «panne byzantine» ou «comportement byzantin» tout comportement d'un système ne respectant pas ses spécifications, en transmettant des résultats non conformes.
On distingue couramment deux types de pannes byzantines :

- Les pannes byzantines naturelles qui proviennent généralement d'erreurs physiques non détectées (mémoire, transmissions réseaux, etc.) ;
- Les pannes byzantines volontaires qui sont issues principalement d'attaques visant à faire échouer le système (sabotage, virus, etc.).

L'authentification et la signature par des moyens de cryptographie permettent de limiter les erreurs byzantines.

Une panne byzantine décrit une situation dans laquelle, pour éviter une panne catastrophique du système, les différents composants de ce dernier doivent s'entendre sur une stratégie concertée, considérant que certains d'entre eux ne sont pas fiables. Ainsi, lorsqu'une panne byzantine se produit dans un composant, tel un serveur, ce dernier apparaît comme étant à la fois en panne et fonctionnel lors de la détection car il présente des symptômes différents aux différents observateurs. Il est donc difficile de le déclarer en panne et de le retirer de l'environnement vu que tous les observateurs doivent d'abord établir un consensus afin de déterminer quel composant est réellement en panne.

On pourrait s'imaginer que dans nos systèmes modernes, il est inutile de considérer ces types de

pannes tellement elles s'avèrent rares ou improbables. On aurait tort car il existe maints exemples de pannes byzantines et de multiples implantations de solutions à celles-ci dans les systèmes modernes :

- Cloudfare (2020)[58] ;
  Le 2 novembre 2020, un commutateur a commencé à se comporter anormalement dans le réseau de l'entreprise Cloudfare. Ce comportement erratique a nui à la disponibilité de certains produits (API et Dashboard) durant plusieurs heures. Pendant cette période, le taux de succès des requêtes a diminué et l'accès à certains environnements fut 80 fois plus lent que la normale.
- La navette spatiale Discovery (vol STS-124) [2] ;
  Le système de contrôle du carburant fut la source de l'échec du vol STS-124 de la navette Discovery. Il y avait quatre systèmes de contrôle. Lors de cet incident, on a découvert que les quatre systèmes produisaient tous des informations différentes et complètement fausses. Le problème se situait toutefois au niveau du matériel (et non logiciel). Il provenait d'une carte de contrôle qui communiquait avec les quatre systèmes en question. Une fissure dans une diode fut à l'origine de cette panne byzantine. Cette situation a provoqué le retard du lancement de la navette.
- Les chaînes de blocs [2] ;
  Des pannes byzantines peuvent se produire dans une chaîne de blocs dues à des nœuds non fiables ou malveillants. Si un membre de la communauté envoie des informations incohérentes à d'autres sur les transactions, la fiabilité de la chaîne de blocs est compromise. Un autre problème, lié aux pannes byzantines, se pose dans cet environnement, celui des doubles dépenses. La double-dépense est une attaque dans laquelle un acteur utilise à deux reprises les mêmes crypto-monnaies. Dans ce cas, le réseau de blocs doit résister à cette attaque par des mécanismes de tolérances aux pannes byzantines partielles ou complètes.
- Les sous-marins [149]
  Des pannes byzantines ont été observées à l'occasion pendant les tests d'endurance des sous-marins de la classe Virginie (jusqu'à 2005).

Quelques exemples d'environnements nécessitant des solutions aux problèmes des pannes byzantines :

- Le Bitcoin et les chaînes de blocs [3, 21]
  Selon Bit2Me [3], l'une des premières solutions pratiques et satisfaisantes pour implanter la tolérances aux pannes byzantines est fournie par le système de cryptomonnaie **BitCoin**, proposé par Satoshi Nakamoto. Depuis, la tolérance à ce type de pannes est généralisée aux systèmes basés sur les chaînes de blocs, dont les cryptomonnaies.
- Les systèmes d'emmagasinage de données [3] ;
  Dans les systèmes permettant d'emmagasiner de grandes quantités de données (système de fichiers ou base de données), différentes techniques de tolérance aux pannes byzantines sont appliquées. Les systèmes de fichiers ZFS sont un exemple d'environnement utilisant ces techniques pour emmagasiner de façon fiable de grandes quantités de données.
- L'aviation [149, 45, 163]
  Plusieurs avions, dont le système de gestion de l'information du Boeing 777 et les systèmes de contrôle de vols des Boeing 777 et 787, font appel à des techniques de tolérances aux pannes byzantins.
- La capsule Dragon de SpaceX [149, 24]
  De même, la capsule spatiale Dragon de SpaceX emploie des techniques de tolérance aux pannes byzantines dans sa conception.

Obtenir un consensus ou un accord

Définition : Consensus

Le «**problème du consensus**» consiste à ce qu'un ensemble de processus s'entendent sur une valeur unique après qu'un ou plusieurs d'entre eux aient proposés des valeurs candidates.

Comme nous venons de l'exposer, obtenir un accord entre processus avant d'effectuer une action est primordial pour une panoplie d'applications présentes dans les systèmes parallèles et distribués. Ainsi, les différentes solutions à certaines problématiques, telle que la synchronisation (exclusion mutuelle, synchronisation d'horloge, ...), l'engagement dans les transactions atomiques et la tolérance aux pannes, nécessitent en général la capacité d'établir un consensus. L'exemple introduit à la figure 7.61 illustre de manière informelle le problème relié à l'obtention d'un consensus en présence de pannes impropres. Le lien avec l'informatique est toutefois facile à établir.

Le problème des quatre armées

L'exemple suivant, inspiré de longues guerres impliquant l'empire byzantin au moyen-âge, illustre bien la difficulté d'arriver à un accord.
Plusieurs généraux du même camp et leur armée sont situés à des extrémités différentes d'une ville avec l'intention de l'assiéger. Pour conquérir la ville, ils se doivent de l'attaquer de manière coordonnée afin de vaincre les défenses ou de battre en retraite de manière coordonnée. Si la synchronisation n'est pas parfaite, ils ne réussiront pas à vaincre les puissantes forces ennemies. Pour se faire, les généraux attaquants communiquent entre eux via des messagers. Bien que la ville regorge d'ennemis, les messagers doivent traverser la ville pour passer d'un camp à l'autre avec les ordres d'attaque contenant en particulier le moment de celle-ci. Lorsqu'un général reçoit un tel ordre, il le confirme ou le rejette afin d'en arriver éventuellement à un accord sur le moment de l'attaque.
Le problème réside dans le fait que les messagers, en traversant la ville, risquent d'être capturés. Le message pourrait alors ne jamais se rendre ou même, être modifié pour empêcher une attaque concertée des armées des généraux (entraînant ainsi leur défaite). Il est même possible d'imaginer qu'un des généraux soit un traître et envoie des messages confus pour corrompre l'accord.
Ainsi, si un message des généraux consiste à «Attaquer demain à 10h», le texte pourrait être modifié par «Attaquer demain à 12h». De cette façon, une armée (ou plusieurs) acceptant la proposition et confirmant le message provoquerait qu'elle seule attaquerait à 12h, tandis que les autres le feraient à 10h.
Un système distribué ressemble à une armée de processus tentant de s'entendre sur un objectif commun. Chaque général correspond à un processus, les messagers au réseau de communication par messages et le traître (ou le faux message) à une panne propre ou impropre du système.
Dans ces conditions, il s'avère difficile voire impossible de coordonner l'attaque des différents camps pour assurer la victoire.

Figure 7.61 – Les problème des quatre armées

Plusieurs environnements existants requièrent l'établissement de consensus pour fonctionner

adéquatement : l'infonuagique, le classement de pages Web (PageRank), les réseaux électriques intelligents, le contrôle de drones (ou d'agents/robots multiples), la répartition de la charge, etc.

Le problème de l'obtention d'un consensus se subdivise en de multiples «variantes» ou en plusieurs sous-problèmes similaires que sont les accords byzantins, le consensus lui-même, la cohérence interactive et quelques autres variations [20, 151] que nous n'abordons pas ici tel l'ordonnancement total des messages dans un système de multidiffusion (multicast).

- Les accords byzantins
  Un accord byzantin exige que tous les processus s'entendent sur une valeur unique alors qu'une valeur initiale, appelée la source, est proposée par un processus désigné. Une solution à ce problème se doit de respecter trois conditions :
  1. l'obtention d'un accord
     Tous les processus valides s'entendent sur une valeur unique.
  2. validité de l'accord ;
     Si la source est valide, tous les processus valides s'entendent sur une valeur unique, i.e. celle proposée par la source.
  3. Terminaison
     Chaque processus valide prend éventuellement une décision sur la valeur finale.
- Le consensus
  Le problème du consensus diffère de l'accord byzantin par le fait que dans ce dernier cas, chacun des processus propose une valeur initiale (il n'y a pas de source unique) et que tous doivent s'entendre sur une unique valeur finale. Les trois même conditions s'appliquent :
  1. l'obtention d'un accord
     Tous les processus valides s'entendent sur la même valeur unique.
  2. validité de l'accord ;
     Si tous les processus valides ont la même valeur initiale, alors l'accord ne concerne que cette seule valeur.
  3. Terminaison
     Chaque processus valide doit éventuellement prendre une décision sur la valeur finale.
- La cohérence interactive
  La cohérence interactive diffère d'un accord byzantin par le fait que chaque processus propose une valeur initiale et que l'accord final concerne un ensemble de valeurs, soit une par processus. Les trois conditions à respecter sont :
  1. l'obtention d'un accord
     Tous les processus valides doivent s'entendre sur le même ensemble de valeurs.
  2. validité de l'accord ;
     Si un processus i est valide et que sa valeur initiale est v_i, alors l'accord concerne v_i en tant que $i^{ième}$ élément de l'ensemble.
  3. Terminaison
     Chaque processus valide doit éventuellement prendre une décision sur l'ensemble des valeurs.

Colouris et al. [20] et Kshemkalyani et Singhal [52] présentent plusieurs solutions aux diverses variantes du problème de consensus. Ces solutions seront éventuellement abordées en détails dans le **cours de systèmes répartis**.

Un problème similaire : la connaissance commune [43, 32, 52]

Le concept de connaissance commune est aussi un concept important pour les systèmes parallèles et distribués. Selon Halpern et Moses [43], ce concept est relié de façon inhérente à la notion d'accord (consensus) et de coordination d'actions. En effet, selon eux, obtenir un accord et coordonner des actions requièrent l'atteinte d'une connaissance commune (c'est un pré-requis !). Cependant, parvenir à cette connaissance commune est ardu et même irréalisable selon le contexte.

Voici quelques cas amusants illustrant cette problématique. L'exemple de la figure 7.62 est une situation dans laquelle deux armées tentent d'obtenir une connaissance commune en présence de communication non fiable. Dans cette situation, le succès de l'opération est fort improbable.

Le problème des deux armées

Soit deux généraux, G1 et G2, alliés dont les armées sont situées à des extrémités différentes d'une ville dans l'intention de l'assiéger. Pour conquérir la ville, il est impératif qu'ils l'attaquent de manière coordonnée pour vaincre les puissantes forces ennemies. Pour y parvenir, les généraux communiquent entre eux via des messagers. Bien que la ville regorge d'ennemis, les messagers doivent la traverser pour passer d'un camp à l'autre pour transmettre les ordres d'attaque. Lorsqu'un général reçoit un message, il confirme sa réception pour établir éventuellement un accord.
Le problème réside dans le fait qu'en traversant la ville les messagers soient possiblement interceptées. Nous simplifions le problème en supposant qu'il ne peut survenir aucune altération au message, seulement une perte.
Examinons maintenant si une entente est possible. Le général G1 envoie un message indiquant à G2 d'attaquer le lendemain à 10h. G2 répond à G1 pour l'assurer de la bonne coordination. Toutefois G2 n'a aucun moyen de vérifier si sa réponse s'est rendue. Le général G1 doit donc répondre pour confirmer qu'il a bien reçu la réponse. Toutefois sa propre réponse peut elle aussi se perdre. Donc G2 devra répondre de nouveau...

Dans ces conditions, combien de messages s'échangeront-ils pour établir une attaque coordonnée ?

Figure 7.62 – Le problème des deux armées

La figure 7.63 elle, illustre la possibilité d'atteindre une connaissance suffisante est possible afin de prendre une décision et ce, même avec une information partielle (évidemment on suppose que les communications sont fiables et que tout le monde est honnête). Celle-ci peut être déduite des connaissances locales.

Il est donc intéressant de constater qu'un certain niveau de connaissance est parfois suffisant pour atteindre un consensus (ce qui n'est pas le cas pour l'exemple des deux armées où une connaissance commune est nécessaire). Plusieurs solutions permettent d'atteindre un niveau de connaissance suffisant pour établir un consensus et s'entendre sur des actions communes. Le cours de systèmes répartis étudiera plus en profondeur ce sujet.

Le problème des enfants sales

Soit n enfants jouant ensemble dans la cour d'école. La personne responsable de la classe leur a bien dit qu'elle les autorisait à s'amuser mais que s'ils revenaient sales, il y aurait de «graves» conséquences. Évidemment chacun d'eux veut donc demeurer propre, mais aimerait voir les autres se salir. Bien sûr, certains d'entre eux, supposons k, reviennent en classe avec de la boue sur le front. Chacun voit la boue sur le front des autres, mais ignore leur propre état. De plus, personne ne parle (évidemment on désire éviter que les coupables aient l'occasion de se laver avant le retour de la responsable)...
La personne responsable de la classe arrive et mentionne :

«*Au moins l'un d'entre vous a de la boue sur son front*».

Elle exprime ainsi un fait que chacun connaissait déjà (évidemment seulement si $k > 1$). La responsable demande alors :

«*Est-ce que certains d'entre vous savez si vous avez de la boue sur votre front ?*»

Sous l'hypothèse que tous les enfants sont perspicaces, intelligents, honnêtes et qu'ils répondent simultanément, que va-t-il se passer ?
Il est possible de prouver que les $k-1$ premières fois que la question sera posée, tous les enfants répondront «non». À la $k^{ième}$ fois, les enfants sales répondront «oui».
Procédons par induction.

- Si $k = 1$, le résultat est évident. L'enfant sale voit qu'aucun autre enfant n'est sale. Il en déduit donc que c'est lui et répond «oui».
- Si $k = 2$, il y a deux enfants sales, supposons a et b. La première fois que la question est posée, ils répondront «non» car chacun voit l'autre enfant sale. Toutefois quand b répond «non», a réalise qu'il doit être sale car autrement b aurait déduit qu'il était le seul à avoir de la boue sur le front et aurait répondu «oui». La seconde fois que la question est posée, a et b répondront donc «oui».
- Si $k = 3$, il y a trois enfants sales, soit a, b et c. L'enfant a réfléchit de la façon suivante. Je suppose que je ne suis pas sale et donc que $k = 2$ (je vois deux enfants sales). Selon le ce cas, b et c devront répondre «oui» la seconde fois que la question est posée. Comme ils répondent «non», a réalise que son hypothèse est fausse et qu'il est sale. Il répondra donc «oui» la troisième fois que la question sera posée (de même pour b et c).
- Le cas général est similaire.

Figure 7.63 – Le problème des enfants sales

7.8 Interblocage

Comme déjà vu, un interblocage est une situation dans laquelle un ensemble de processus sont bloqués puisque chacun d'entre eux détient une ressource mais est en attente d'une autre ressource qui elle, est détenue par un autre processus. Cette situation a été abordée dans le cours IFT320 dans le cadre de système centralisé.

Interblocage - analogie [91, 160]

Loi passée au Kansas au début du $20^{ième}$ siècle

> When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone.

Interblocage - analogie [91]

Catch-22

> Terme utilisé en référence au roman de Joseph Heller pour désigner une situation paradoxale dans laquelle un individu ne peut éviter un problème en raison de contradictions dans les règles ou les contraintes sur lesquelles on a aucun contrôle. C'est donc une situation sans issue (damned if I do, damned if I don't).)
>
> Quelques exemples de Catch-22 :
>
> - Vous ne pouvez pas obtenir un emploi sans avoir de l'expérience, mais vous ne pouvez pas accumuler de l'expérience sans avoir un emploi.
> - L'imprimante au bureau n'a plus d'encre. Il faut compléter et imprimer un formulaire pour demander de l'encre, mais ... il n'y a pas d'encre pour l'imprimer [164].
> - «In quantum computing, you must observe a particle in order to determine its location. However, the very act of observation affects the quantum behavior (and location) of particles.[164]»
> - «You want to know whether the majority of internet users desire a higher level of privacy. In order to do this, you must collect data about the users.» [164]

Les interblocages sur les systèmes distribués sont similaires à la différence qu'ils sont plus complexes à **éviter**, à **prévenir**, à **détecter** et à **corriger**. En effet, comme l'information est disséminée sur plusieurs machines et qu'aucune d'entre elles ne possède une connaissance précise de l'état du système (dû aux délais de communication), il s'avère relativement ardu de regrouper ces informations de manière cohérente afin d'en déduire une conclusion pertinente.

Sur un système distribués, deux types d'interblocages sont définis :

- les interblocages de ressources ;
  Ce type d'interblocage (celui traité au cours IFT320) se produit quand deux ou plusieurs

processus attendent pour une ressource qui ne se libérera jamais étant elle-même détenue par un autre processus (qui lui aussi attend).

Par exemple, un processus P_1 détient les ressources R_1 et R_2 et demande la ressource R_3. Il ne s'exécutera pas tant qu'il ne détiendra pas cette dernière ressource. Cependant si un processus P_2 détient R_3 et demande une des deux ressources R_1 ou R_2, nous obtenons un interblocage (P_1 attend après P_2 qui lui, attend après P_1.)

- les interblocage de communication

  Un interblocage de communication se produit à même un ensemble de processus lorsque certains d'entre eux sont bloqués en attente de messages censés provenir de d'autres processus (du même ensemble) mais ce en vain puisque les dits messages n'ont pas été transmis (les messages ne sont pas en transit). Cela signifie, par exemple, qu'un processus P_1 attend un message d'un processus P_2, qui attend un message d'un processus P_3, qui attend un message de P_1.

  Les interblocages de communications se modélisent aisément de la même façon que les interblocages de ressources, i.e. avec des graphes d'allocation de ressources, et donc se traitent aussi de la même façon.

La plupart des techniques déjà considérées pour traiter les interblocages s'adaptent aux systèmes distribués.

7.8.1 Ignorer le problème (L'algorithme de Ostrich)

Il est toujours possible de supposer qu'il n'y aura jamais d'interblocages et d'ignorer le problème. Il va s'en dire que cette méthode s'adapte très bien à n'importe quel environnement (centralisé, parallèle ou distribué).

7.8.2 Horloge de garde

Des horloges de garde (timeout) permettent de fixer une limite de temps sur la détention d'un verrou. Ainsi avant ce délai de grâce, le verrou est invulnérable. Passé ce délai, le verrou devient vulnérable, c'est-à-dire que si un autre processus demande la ressource, le verrou sera levé et le détenteur détruit. Si aucun processus ne désire le verrou, le détenteur le conservera et poursuivra son exécution.

Utiliser des horloges de garde pour résoudre les interblocages n'est pas vraiment une bonne approche car il est difficile de déterminer l'intervalle de temps adéquat et certains processus risquent d'être annulés inutilement.

7.8.3 Prévention des interblocages

La prévention des interblocages consiste à concevoir le système de façon à rendre les interblocages structurellement impossibles. Plusieurs techniques existent telles que permettre aux processus de détenir une seule ressource à la fois, les obliger à demander toutes les ressources au début de leur exécution ou leur demander de relâcher toutes leurs ressources lors d'une nouvelle demande. Cependant toutes ces approches se révèlent assez peu pratiques et entraînent éventuellement des ralentissements ou des blocages inutiles. Une autre approche vise à ordonnancer globalement les ressources et à exiger qu'elles soient acquises dans un ordre croissant.

Toutefois avec un système distribué d'autres approches sont envisageables dont deux en particulier sont basées sur l'utilisation d'estampilles. Chaque processus (ou transaction lancée par un processus) se voit alors assigner au début de son exécution une estampille unique. Décrivons ces deux algorithmes :

- **algorithme sans réquisition avec attente ou annulation (wait-die) ;**
  Lorsqu'un processus P_i tente d'acquérir une ressource déjà occupée par un processus P_j, les estampilles sont comparées et P_i est autorisé à attendre que P_j libère la ressource seulement si son estampille est plus petite (demande plus ancienne). Dans le cas contraire, le processus est détruit (et ré-initialisé avec une nouvelle estampille). Cette approche fait en sorte que les estampilles des processus en attente (dans une file) sont toujours croissantes. Les cycles sont donc impossibles.

  L'alternative inverse est aussi concevable, i.e. P_i attend que P_j libère la ressource seulement si son estampille est plus grande (la demande est plus récente). Dans ce cas, les estampilles sont décroissantes dans la file d'attente.

  Même si les deux approches fonctionnent, il est préférable de donner priorité aux processus plus anciens. En effet, comme ils s'exécutent depuis une plus longue période de temps, ils détiennent potentiellement plus de ressources et le système a «un plus grand investissement» en ceux-ci. De plus, quand un processus plus jeune est détruit, il aura éventuellement la chance de vieillir ce qui élimine les risques de la famine.
- **algorithme avec réquisition basée sur l'attente ou réquisition/annulation (wound wait).**
  Lorsqu'un processus P_i demande une ressource détenue par le processus P_j, les estampilles sont comparées et P_i attend que P_j libère la ressource si son estampille est plus grande (demande plus récente). Si l'estampille de P_i est plus petite, le processus P_i réquisitionne la ressource et le processus P_j est détruit (et ré-initialisé). Pour que cette algorithme fonctionne, il est absolument nécessaire que les processus utilisent des transactions qui permettront d'annuler toutes les modifications déjà faites par P_j.

7.8.4 Évitement

L'évitement des interblocages consiste à allouer soigneusement les ressources de façon à ce que l'interblocage ne puisse se produire. L'algorithme du Banquier est un exemple d'une telle approche proposée pour les systèmes centralisés.

Toutefois, selon Tanenbaum [103] et Ksemkalyani et Singhal [52], cette approche est rarement utilisée et même impraticable dans les systèmes distribuées. En fait, selon Tanenbaum [103], elle ne l'est même pas pour les systèmes centralisés. La difficulté de l'algorithme du Banquier et autres algorithmes similaires réside dans la nécessité de connaître à l'avance le nombre de ressources requis par chacun des processus. Évidemment, cette information est rarement disponible (ou jamais).

Malgré tout, certains travaux [40, 83, 60, 79] ont tenté de développer des algorithmes d'évitement pour les systèmes distribués. Son principe est le même que celui de l'algorithme du Banquier, i.e. que cela consiste à allouer les ressources demandées seulement si l'état global est «sûr». Dans le cas contraire (un état global «non-sûr»), un interblocage serait à craindre si on procédait à l'allocation, il faut donc l'éviter. Certains travaux [40] ont mis en évidence la complexité et la lenteur avec lesquelles on parvient à obtenir un état global et à valider la demande. Elles sont telles que l'application de ces techniques est à peu près impossible. Pour régler ce problème d'autres études [83, 60] ont permis

de développer des algorithmes basés uniquement sur de l'information locale afin de décider ou non de l'allocation.

7.8.5 Détection et reprise

Comme c'est le cas sur les systèmes centralisés, il est aussi possible de détecter un interblocage sur les systèmes distribués. Il existe quatre approches pour y parvenir :

- **Détection centralisée grâce au graphe global d'allocation des ressources («wait-for graph») ;**
  Selon cette approche, chaque site construit un graphe d'allocation local. Si un cycle est présent dans le graphe local la détection est immédiate. Le graphe global est ensuite construit à partir de ces graphes locaux afin qu'un «processus coordonnateur» puisse agir. Plus de détails dans les sections suivantes.
- **Détection décentralisée par l'envoi de chemins (path-pushing) ;**
  Selon cette approche chaque site construit un graphe d'allocation global et détecte des interblocages. Quand un site doit exécuter un algorithme de détection, il envoie son graphe local à tous ses voisins. Quand le graphe local de chaque site voisin est mis à jour, ils transmettent à leur tour le graphe à jour à tous leurs voisins. Cette procédure est répétée jusqu'à ce qu'un site possède suffisamment d'informations sur l'état global pour établir la présence ou non d'un interblocage.
- **Détection décentralisée par la chasse aux arcs (edge chasing) ;**
  Cette approche consiste à rechercher les cycles dans un graphe distribué sur plusieurs sites par l'envoi de messages spéciaux appelés «sondes». Plus de détails dans les sections suivantes.
- **Détection décentralisée par diffusion du calcul.**
  Selon cet algorithme, le traitement de la détection est diffusé sur le graphe d'allocation. Aucun graphe global n'est explicitement construit. Il fonctionne d'une façon similaire à l'algorithme de chasse aux arcs.
  Pour détecter un interblocage, un processus P_1, appelé l'initiateur, diffuse un message de type «requête» sur tous les arcs sortant du graphe d'allocation des ressources (toutes les demandes non accordées). Un processus P_2 recevant une requête de P_1 l'ignore s'il s'exécute. S'il est bloqué en attente d'une ou plusieurs ressources, il envoie des messages de type requête à tous ses successeurs dans le graphe (tous les arcs sortant). Pour toutes les autres requêtes concernant la demande de détection de P_1, P_2 répondra sans attendre. Finalement, P_2 enverra une réponse à P_1 lorsqu'il aura reçu des réponses de tous ses successeurs. P_1 détecte un interblocage lorsqu'il a reçu une réponse à toutes ces requêtes.

Détection centralisée avec un graphe global

Selon cet algorithme, un coordinateur se charge d'identifier les interblocages (en détectant les cycles dans le graphe) suite à l'élaboration d'un graphe d'allocation global (union des graphes locaux).

La construction du graphe global est cependant une tâche non triviale. En effet, l'information est distribuée sur plusieurs sites et ceux-ci transmettent leur «graphe d'allocation local» au coordonnateur. Cela peut sembler simple : chaque site construit son graphe d'allocation local puis le transmet au coordonnateur selon une des approches suivantes :

- à chaque ajout ou retrait d'un arc, un message est envoyé au coordonnateur pour indiquer cette modification ;

- périodiquement, chaque processus envoie au coordonnateur une liste des arcs ajoutés ou retirés depuis la dernière mise à jour ;
- le coordonnateur demande l'information requise lorsqu'il en a besoin.

Malheureusement ces approches simplistes ne fonctionnent pas. En effet, elles entraînent régulièrement la détection de faux cycles (des interblocages fantômes), principalement causés par les délais de communication.

Reprenons l'exemple suivant tiré du manuel de Tanenbaum [103]. Soit deux processus A et B s'exécutant sur un ordinateur 0 et un processus C s'exécutant sur l'ordinateur 1. Ces processus se partagent trois ressources : R_1, R_2 et R_3. Initialement, nous avons la situation suivante (illustrée à la figure 7.64 au temps T_1) : A détient R_2 et demande R_1 elle-même détenue par B. C détient R_3 et demande R_2 (détenue par A). La «vue» du coordonnateur au temps T_1 lui indique qu'il n'y a aucun interblocage (aucun cycle dans le graphe). Aussitôt que B finira, A pourra s'exécuter et terminer, ce qui permettra à C de s'exécuter.

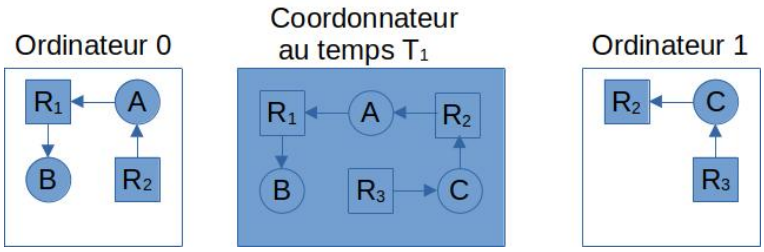

Figure 7.64 – Exemple de détection d'un interblocage fantôme

Au temps T_2, le processus B demande R_3 et le processus C libère R_3. L'ordinateur 0 envoie un message au coordonnateur lui indiquant la demande pour R_3 et l'ordinateur 1 envoie un message au coordonnateur annonçant la libération de R_3. Si l'arrivée du message en provenance de l'ordinateur 1 est retardée, le coordonnateur découvrira un cycle dans le graphe, croira faussement qu'il y a un interblocage et détruira un des processus (figure 7.65 au temps T_2).

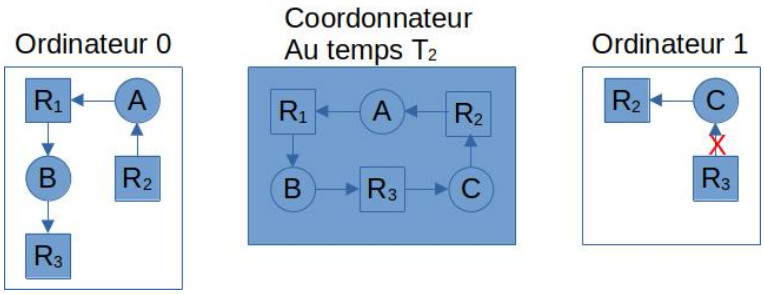

Figure 7.65 – Exemple de détection d'un interblocage fantôme

Pour construire une graphe global et s'assurer qu'aucun interblocage fantôme ne se produise, on s'appuie sur différentes techniques dont l'une d'elles consiste à faire appel aux horloges logiques

(estampilles). Une autre se sert d'un algorithme permettant de construire un état global cohérent [20, 52]. Ce dernier sujet sera à l'étude dans le cours de systèmes répartis.

Détection décentralisée avec la chasse aux arcs

Une détection centralisée pose différents problèmes dont le principal est la fiabilité. Plusieurs algorithmes distribués ont été développés pour détecter les interblocages dont plusieurs sont basés sur des techniques appelées la «**chasse aux arcs**» (*edge chasing*).

Selon cette approche, aucun graphe d'allocation global n'est construit. Cependant chaque site impliqué connaît une partie du graphe. Un processus tente de découvrir un cycle en envoyant un message de type «sonde» qui suit les arcs du graphe à travers tout le système distribué. La sonde voyage sur le chemin d'une demande. Un tel chemin est localisé entre deux processus A et B si A demande une ressource détenue par B et que ce dernier est bloqué. Ainsi le processus A envoie la sonde à B.

Un tel algorithme a été développé par Chandy, Misra et Hass [15]. Selon leur algorithme, un message sonde est généré lorsqu'un processus doit attendre pour une ressource et envoyé au processus détenant la ressource. Quand un message sonde arrive (à un processus détenant une ressource), le récepteur vérifie s'il attend lui-même pour une ressource. Si oui, la sonde est mise à jour et expédiée au processus détenant la ressource sur laquelle il est bloqué. S'il attend plusieurs ressources, une sonde est envoyée à tous les processus. Si le récepteur n'est pas bloqué, il détruit la sonde. Si un message revient à l'expéditeur original, c'est l'indication qu'un cycle existe et qu'il y a un interblocage.

Soit l'exemple de graphe d'allocation illustré à la figure 7.66 (inspiré de [103, 20]). Supposons que le processus P_0 demande la ressource R_1. Comme R_1 est déjà occupé par P_3, P_1 envoie un message sonde à P_3 contenant trois valeurs : le première représentant le processus venant de se bloquer, le second le processus qui envoie le message et le troisième le processus destinataire. Comme le processus P_3 est bloqué sur une ressource occupée par P_4, il met à jour le message sonde et l'envoie à P_4, Finalement comme P_4 est aussi bloqué, il transfère le message sonde vers P_0 après avoir procédé à sa mise à jour. P_0 s'aperçoit alors qu'il est l'initiateur du message et détecte donc un interblocage. Il agira alors pour éliminer l'interblocage.

7.9 Autres concepts importants

Il existe plusieurs autres concepts reliés aux systèmes parallèles et distribués et ceux-ci devraient être abordés au cours de systèmes répartis, dont :

- **Les états globaux ;**
  Un état global est la collection de tous les états locaux des processus participants au traitement. Ils sont très utiles pour la détection d'interblocage, la détection de terminaison d'algorithme, la mise au point distribué et les ramasses-miettes distribués. Toutefois, l'obtention d'un état global cohérent se révèle très complexe dû aux retards dans la livraison de l'information qu'imposent les délais de communication. Un chapitre complet du manuel de Kshemkalyani et Singhal [52] est consacré à la capture d'états globaux cohérents.
- **La mise au point de programmes parallèles et distribués ;**
  Cette mise au point est basée sur les états globaux et la journalisation d'un exécution.
- **Protection/identification/authentification/sécurité ;**

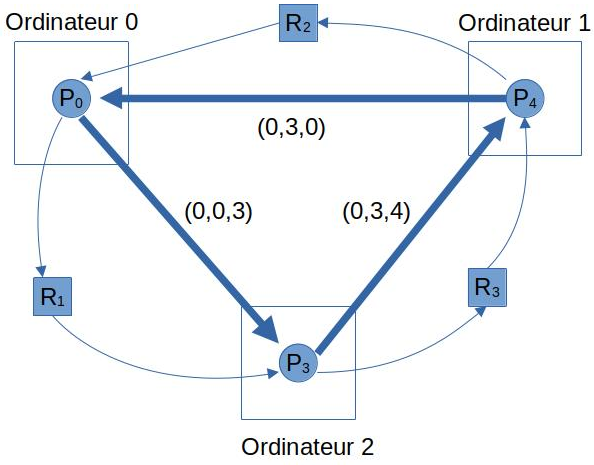

Figure 7.66 – Exemple de détection d'un interblocage fantôme

Puisqu'un calcul parallèle peut s'étendre sur plusieurs ordinateurs, les problèmes d'identification, d'authentification et de protection deviennent plus complexes.

- **Hétérogénéité ;**
  Si un calcul s'exécute sur un réseau d'ordinateurs potentiellement hétérogènes, il faudra tenir compte de ce fait. Par hétérogénéité, on signifie des différences au niveau :
  - du matériel (Intel vs Arm vs Risc V vs ...) ;
  - du logiciel (Windows vs MacOS vs Linux vs ...) ;
  - de la configuration (quantité de mémoire ou de disques, vitesse de l'UCT, ...) ;
  - des versions.

  Dans tous les cas, il faut en quelque sorte «s'immuniser» contre ses différences. Pour y parvenir diverses solutions sont proposées :
  - des formats de données «universels» ;
    XDR est une tentative de standardisation des données. MPI utilise son propre format de données pour régler ces problèmes.
  - des intergiciels (ou middelware) ;
    Ces logiciels servent à traduire l'information et cacher les différences matérielles et logicielles. MPI et OpenCL sont des intergiciels tentant de cacher partiellement l'hétérogénéité.
  - des langages de haut niveau ou de script (Java, Python, C#, ...) ;
    Ces langages utilisent leur propre machine virtuelle (JVM, .Net, ...) permettant ainsi de camoufler l'hétérogénéité sous jacente.
- Autres ? ? (à venir).

Annexe A

Raid et mémoire stable

Pour obtenir une unité d'emmagasinage d'information fiable, on fait fréquemment appel aux concepts de RAID et/ou de mémoire stable. Brièvement, un RAID (*Redundant Array of Independant Disks*) est un regroupement de disques physiques alors que la mémoire stable est un concept logique qui assure l'atomicité des opérations et la cohérence/disponibilité des données.

A.1 RAID

La technologie appelée RAID [86, 4, 121, 77, 48, 27, 100, 104, 90] permet d'obtenir, à partir du regroupement de plusieurs disques physiques, un seul disque logique plus performant et plus fiable. Le principe de base le plus important d'un RAID est sa transparence car l'usager doit considérer le regroupement comme un seul et unique disque (logique).

Historiquement, le concept de RAID a été introduit dans les années 80 pour obtenir, et ce à faible coût, un disque logique d'une plus grande capacité, un temps d'accès plus court, un meilleur taux de transfert et une disponibilité accrue, par rapport aux disques physiques uniques. À l'époque l'acronyme RAID signifiait «*Redundant Array of Inexpensive Disks*».

Aujourd'hui, la capacité des disques s'étant significativement améliorée, cette technologie sert surtout à accroître le temps d'accès, le taux de transfert et la disponibilité. Même si le coût des disques est moins important, on utilise fréquemment des disques peu coûteux aux performances moyennes pour construire des disques logiques selon l'architecture RAID.

Les principes sous-jacents à la technologie RAID sont de :

1. considérer qu'un ensemble de disques constitue une seule et unique entité logique ;
2. répartir les données (*striping*) sur tous les disques physiques afin de fournir de meilleurs temps d'accès et d'augmenter le taux de transfert ;

   La répartition des donnés est effectuée en subdivisant les agrégats de données par bandes (agrégat par bandes). Une bande est un regroupement de données sur un disque. Ce peut être des bits, un secteur, un bloc (regroupement de secteurs), une piste, etc. Toutes les bandes d'un même agrégat sont positionnées au même endroit sur les disques physiques. Comme elles

se situent sur des disques physiques distincts, elles sont lues et écrites en parallèle. Ainsi, en théorie, si un RAID contient N disques physiques, le taux de transfert pourrait être N fois plus rapide que sur un disque physique unique.

3. se servir d'informations redondantes pour augmenter la fiabilité (haute disponibilité des données).

   La redondance sur un RAID est conçue pour être capable, en cas de panne, de régénérer les données manquantes à partir des données encore disponibles et de l'information redondante. Cette dernière peut toutefois prendre plusieurs formes. Les plus courantes sont les copies multiples et les bits de parité.

   Il y a une différence entre la redondance appliquée sur un disque physique unique et celle appliquée sur un RAID. Sur un disque unique, la redondance prend généralement la forme d'un code CRC emmagasiné à la fin de chaque enregistrement (sur le même disque). Dans le cas d'un RAID, l'information redondante est emmagasinée sur des disques supplémentaires. Ainsi, en cas de panne d'un disque, il est possible de récupérer les informations (ce qui n'est pas le cas sur un disque unique même avec un code CRC). De plus, sur un RAID, l'écriture de l'information redondante ne ralentit pas l'accès puisqu'elle se fait en parallèle.

Le RAID s'oppose fréquemment au SLED (*Single Large Expensive Disk*). Ce dernier est un disque plus coûteux car il offre une plus grande capacité d'emmagasinage et une meilleure fiabilité. Toutefois si le disque tombe en panne, on perd l'accès à toutes les données, alors que sur un RAID, on peut toujours accéder aux données restantes et régénérer les données en utilisant la redondance.

Il existe de multiples façons de combiner entrelacement des données et redondance pour obtenir une architecture RAID. L'industrie s'est donc entendue sur certains standards pour l'entrelacement et la redondance que l'on appelle les niveaux RAID. Le niveaux les plus populaires sont le RAID 5 et des hybrides de RAID 0 et RAID 1. Nous allons donc présenter les niveaux RAID dans les prochaines sections.

A.1.1 RAID 0 - Entrelacement par bandes

Le niveau 0 des architectures RAID est simplement une agrégation par bandes (entrelacement par bandes ou «*striping*»).

Cette architecture requiert au moins deux disques physiques. Selon cette organisation, un fichier est divisé en N bandes, une bande étant idéalement une combinaison de plusieurs secteurs. Si notre disque logique RAID possède K disques, alors K bandes seront écrites sur les disques en parallèle. Cette situation est illustrée à la figure A.1. Dans cet exemple, le fichier est divisé en 16 bandes réparties sur quatre disques. Supposons que la taille des bandes est de 32 K. Les 16 bandes contiendraient alors un fichier dont la taille serait de 500K ($500 \div 32 = 15,625 \rightarrow 16$ bandes)

L'efficacité de cette approche dépend des demandes d'entrées/sorties. Si celles-ci impliquent le transfert d'une grande quantité de données à la fois, alors on gagne en vitesse. Toutefois si elles ne transfèrent que de des petites quantités de données à la fois, par exemple un secteur, alors le gain est minimal sinon nul. Ce type d'organisation est surtout utile pour les applications ayant besoin d'une vitesse de transfert élevée telle que celle requise par les applications de montage vidéo.

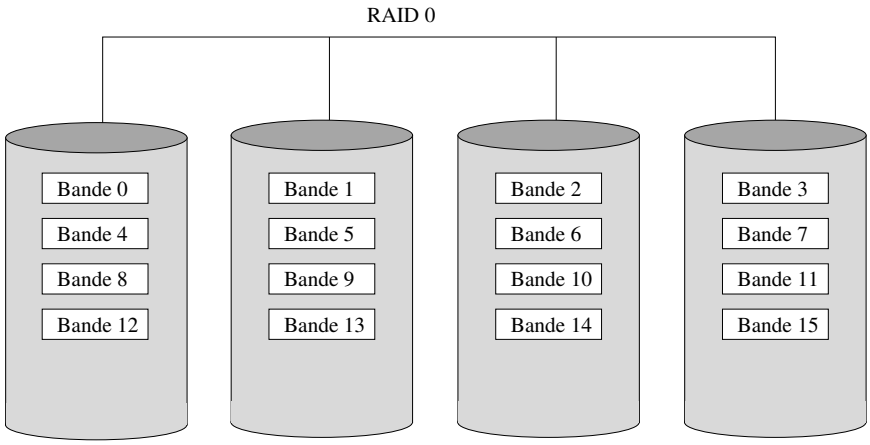

Figure A.1 – Architecture RAID de niveau 0

L'espace total disponible sur une telle architecture dépend de celui du plus petit disque. Ainsi, pour K disques, dont le plus petit est de 500G, la capacité totale est de $K \times 500$G. Donc pour une architecture RAID 0 de deux disques de 500G, son volume total sera de 1000G. Mais, si une architecture RAID 0 se compose d'un disque de 500G et d'un autre de 1000G, sa capacité totale reste de 1000G car l'espace excédentaire du second disque ne sera jamais utilisé. La figure A.2 illustre ce problème de perte d'espace.

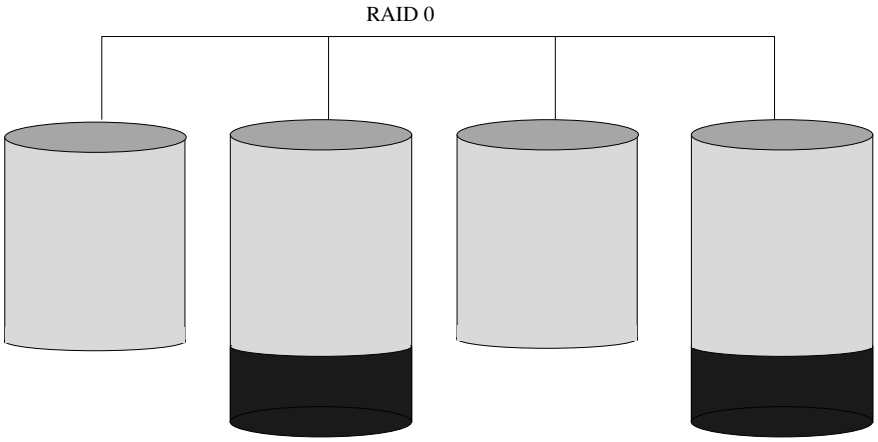

Figure A.2 – Perte d'espace potentielle avec RAID 0

L'inconvénient de cette approche est qu'elle n'offre aucune fiabilité. En effet, s'il y a panne d'un disque, les données deviennent inaccessibles. Si ce disque devient illisible, les données qu'il contenait ne pourront pas être récupérées (sauf si on a une copie de sécurité). En fait, sa fiabilité est moindre que celle d'un «SLED» et même, moindre que celle d'un seul disque (plus grand est le nombre de

disques, plus la probabilité de pannes augmente).

Pour tout dire, l'arrivée des disques SSD semble avoir sonné le glas de cette architecture.

A.1.2 RAID 1 - Disques miroirs

L'architecture RAID de niveau 1 emploie K disques redondants. Selon cette organisation, tous les disques contiennent exactement les mêmes données, d'où le terme miroir. La figure A.3 décrit une telle architecture.

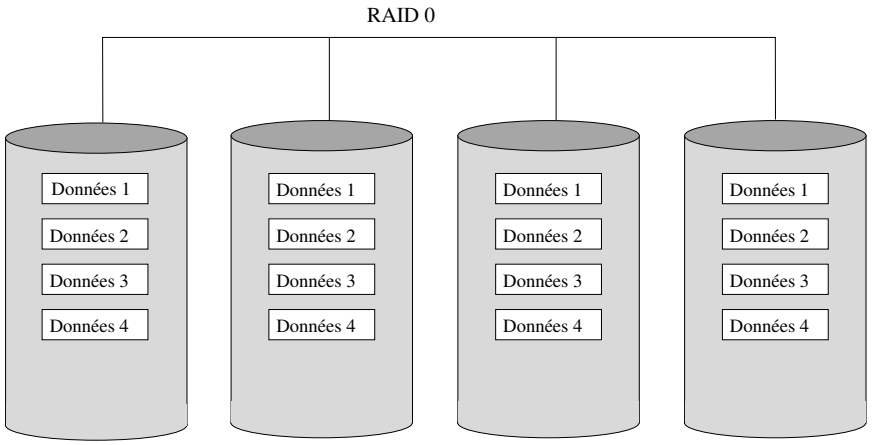

Figure A.3 – Architecture RAID de niveau 1 - Disques miroirs RAID 0

Cette architecture apporte un grande fiabilité mais peu d'accélération concernant l'accès aux données. Les quelques améliorations en performance proviennent du fait qu'une demande d'entrées/sorties a l'avantage d'être dirigée vers le disque physique présentant la plus petite latence (déplacement de la tête de lecture et plus petite rotation du disque)

Le coût de la mise à jour des multiples copies est nul car toutes le écritures sont exécutées en parallèle sur tous les disques en même temps. La reprise après une panne est aussi très simple selon cette approche.

Les inconvénients de cette architecture sont principalement le manque d'accélération lors du traitement des demandes et son coût très élevé (en terme de disques). Pour palier ce problème, on combine généralement les architectures de niveau 0 et 1. On obtient alors une accélération pour l'accès et une fiabilité accrue en cas de panne. Cette architecture mixte est présentée à la figure A.4.

A.1.3 RAID 2 - Code correcteur d'erreurs

L'architecture RAID de niveau 2 applique des techniques de détection et de correction d'erreurs identiques à celles employées pour la mémoire centrale des ordinateurs. Elle fait appel à une technique d'entrelacement par bandes (RAID 0) combinée à une méthode de détection d'erreurs

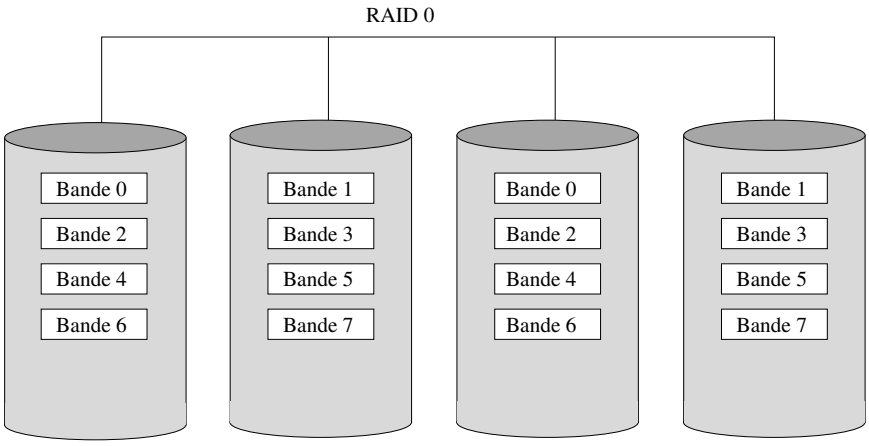

Figure A.4 – Architecture RAID mixte combinant les niveaux 0 et 1

afin d'obtenir une unité rapide et fiable. Notons que le code de Hamming est souvent la méthode privilégiée pour le contrôle des erreurs.

Ce type d'architecture favorise fréquemment l'entrelacement par bandes de 1 bit. L'unité de répartition est très petite, soit un octet ou un mot. Les bandes (de 1 bit chacune) de l'unité de répartition sont emmagasinées sur des disques distincts du RAID et les bandes de contrôle (bits de contrôle), elles, le sont sur d'autres disques. La bande de données en position i contient le $i^{ème}$ bit de chaque unité tandis que les bandes de contrôle contiennent chacune un des bits de contrôle.

Par exemple, avec un code de Hamming appliqué à un octet, on obtient 4 bits de données et 3 bits de contrôle. Sept disques sont donc nécessaires pour emmagasiner tous les bits. Il est aussi possible d'utiliser 8 disques de données (8 bits) pour 4 bits de contrôle. L'ordinateur CM-2, un ordinateur expérimental, employait 32 bits de données (32 disques), 6 bits de contrôle basés sur le code de Hamming (6 disques) et un bit de parité (un autre disque). Le RAID comprenait donc 39 disques.

Le niveau RAID 2 est n'est guère usité de nos jours car il est complexe, coûteux et désormais désuet. En effet, il propose un contrôle d'erreurs qui est maintenant directement intégré dans les contrôleurs de disques durs.

A.1.4 RAID 3 - Entrelacement par bandes (octets) et bits de parité

L'architecture RAID de niveau 3 emmagasine les informations avec un entrelacement par très petites bandes (un octet) sur plusieurs disques. De plus, elle nécessite un disque supplémentaire pour emmagasiner des bits de parité. Un RAID de niveau 3 comprend donc K disques, $K-1$ pour les données et un pour la redondance. La figure A.5 illustre une architecture RAID de niveau 3 contenant 4 disques.

Cette architecture offre une très grande rapidité d'accès, un haut taux de transfert, la fiabilité et cela à un coût moindre que les architecture précédentes (RAID 1 et RAID 2). En effet, elle ne

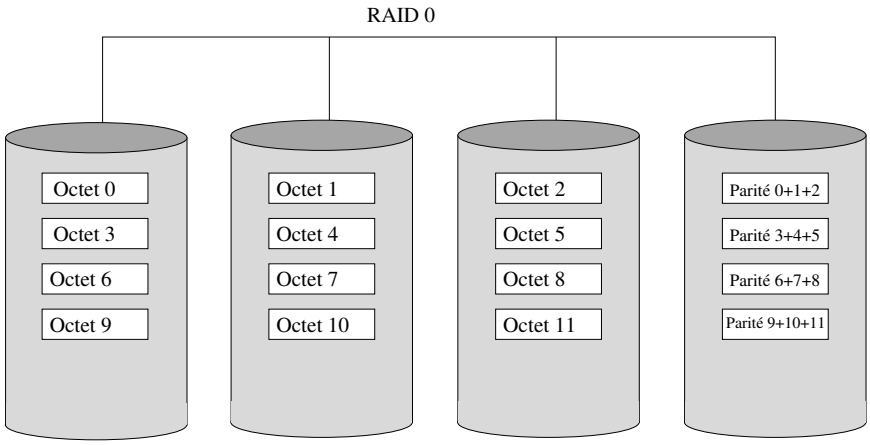

Figure A.5 – Architecture RAID de niveau 3

nécessite toujours qu'un seul disque de redondance et ce, peu importe le nombre de disques de données.

La fiabilité est obtenue par les bits de parité. Ainsi, si un disque subit une défaillance, il est possible de reconstruire aisément l'information à partir des autres disques. Soit un système RAID 3 contenant 5 disques : 4 pour les données (X_0, X_1, X_2 et X_3) et un pour la redondance (X_4). Le bit de parité pour les bits en position i pour toutes les bandes à travers tous les disques est obtenu grâce à la formule suivante :

$$X_4[i] = X_3[i] \oplus X_2[i] \oplus X_1[i] \oplus X_0[i]\,^1$$

Si l'un des disques tombe en panne (supposons X_1), la récupération du bit en position i pourra se faire par la calcul suivant :

$$X_1[i] = X_4[i] \oplus X_3[i] \oplus X_2[i] \oplus X_0[i]$$

Ainsi, tout le contenu initial du disque X_1 sera régénéré. Par contre, cette architecture ne tolère qu'une seule panne.

La rapidité d'accès est assurée ici par la lecture en parallèle de toutes les bandes de données (haut taux de transfert). Comme celles-ci sont petites, les données seront systématiquement réparties sur la totalité des bandes. Toutefois ce fait implique que tous les disques doivent être synchronisés en permanence et participent tous à chaque demande d'entrées/sorties. Cet avantage s'accompagne de l'inconvénient : le traitement d'une seule demande à la fois (moins d'entrées/sorties par seconde).

Notons en terminant que les écritures sur disque subissent un certain ralentissement dû au calcul de parité. Par contre, comme tous les disques sont synchronisés en permanence, l'écriture elle-même sur les disques ne souffre d'aucun délai.

1. Le symbole $\oplus$ représente un «ou exclusif»

A.1.5 RAID 4 - Entrelacement par bandes (blocs) et bits de parité

L'architecture RAID de niveau 4 est similaire à celle du RAID 3 sauf qu'il privilégie un entrelacement par bandes dont la taille est beaucoup plus grande, d'au moins un secteur. Ainsi, un RAID 4 de K disques, emmagasinera toutes les bandes de données (blocs) sur $K-1$ disques et la bande de contrôle (bits de parité) sur le dernier disque. La parité est calculée de la même façon que pour le RAID 3 sauf qu'elle s'applique sur des bandes de tailles supérieures. La figure A.6 décrit l'architecture du RAID niveau 4.

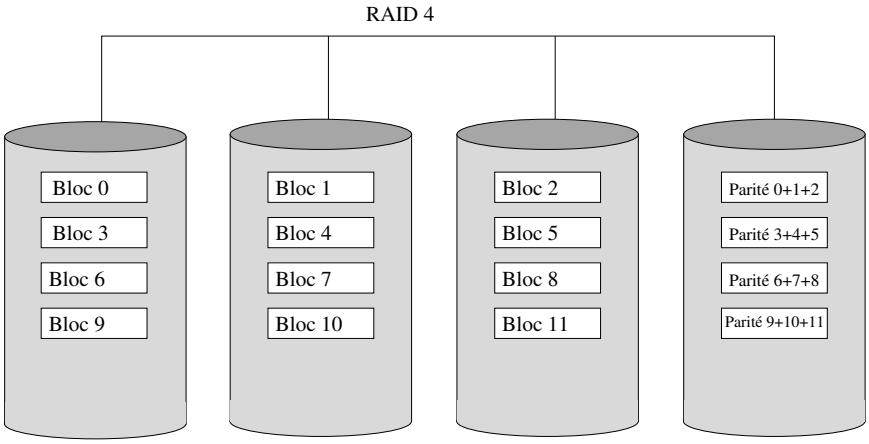

Figure A.6 – Architecture RAID de niveau 4

Comme le RAID 3, cette architecture offre la rapidité d'accès, un haut taux de transfert et la fiabilité (au même coût que le RAID 3). Le RAID 4 est cependant plus performant dû principalement à la taille de ses blocs (par rapport aux octets). Ainsi, lors de l'écriture de grandes quantités de données, celles-ci peuvent être emmagasinées en parallèle sur tous les disques. Cependant, contrairement au RAID 3, il n'est pas nécessaire de synchroniser tous les disques en permanence. En effet, les bandes (blocs) étant de grandes tailles, elles peuvent être indépendantes (i.e. les lectures ou écritures n'ont pas à accéder à toutes les bandes sur tous les disques en même temps pour manipuler les données). Ainsi, lors de l'écriture d'une quantité relativement faible d'informations (qui entrent dans une seule bande), plusieurs demandes d'E/S pourront être acheminées concurremment et ce, tant qu'elles opèrent sur des disques distincts (plus d'entrées/sorties par seconde).

Comme le RAID 3, cette approche souffre d'une certaine pénalité à l'écriture due au calcul de parité. Toutefois, dues à l'indépendance des blocs (les disques ne sont pas synchronisés), lors de la mise à jour d'une bande unique, la lecture et l'écriture de la bande de parité associée sont requises, afin de recalculer les nouveaux bits de parité. Donc une mise à jour, même mineure, exige deux lectures et deux écritures (bande de données et bande de parité).

A.1.6 RAID 5

En ce qui concerne l'architecture RAID de niveau 5, c'est la même que celle du RAID de niveau 4 à l'exception que dans cette version, les bandes de contrôle (bits de parités) sont réparties sur tous les disques. Une architecture de type RAID 5 de K disques, utilise $K-1$ bandes de données réparties sur les K disques et une bande de contrôle (aussi répartie sur les K disques). La figure A.7 illustre une architecture RAID 5 de 4 disques.

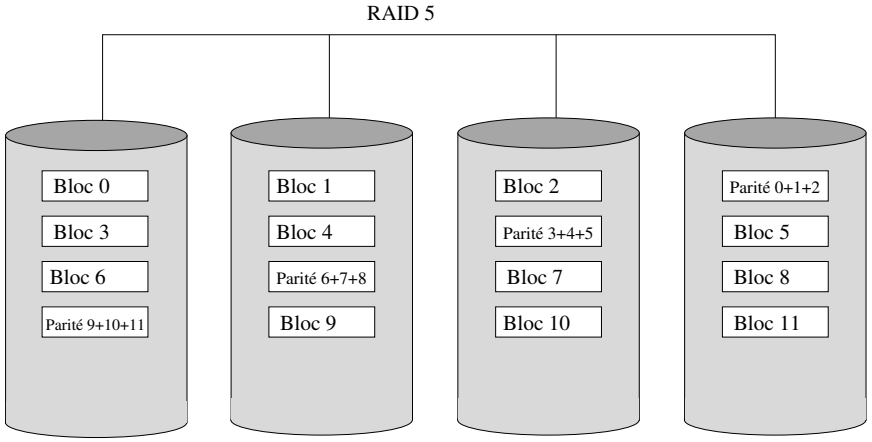

Figure A.7 – Architecture RAID de niveau 5

Cette architecture est plus efficace que celle du RAID 4 car, dans ce dernier, le disque de parité est susceptible de devenir un goulot d'étranglement. Cette situation ne peut survenir sur un RAID 5 car les écritures des informations de contrôle (bits de parité) sont mieux distribuées. De plus, cet avantage est obtenu sans coût supplémentaire.

Ce type d'architecture est l'un des plus populaires.

A.1.7 RAID 6

L'architecture RAID de niveau 6 utilise une double parité. C'est cette seule particularité qui la distingue de la version précédente, RAID 5. Ce fait permet de tolérer des pannes sur deux disques. Cette architecture exige un minimum de quatre disques (deux disques de données et deux disques de parité). Les deux bandes de contrôle nécessitent des techniques de calcul de parité différentes. Soit P et Q, ces parités respectives. La parité P peut être obtenue par un simple «ou exclusif». Quant à la parité Q, on le calcule à l'aide d'un autre algorithme tel que le code de Hamming ou le code de Reed-Solomon. La figure A.8 expose une architecture RAID 6 de 5 disques.

Cette architecture est plus fiable que les autres versions RAID mais plus lente que le RAID 5 étant donné les calculs supplémentaires de parité qui doivent être effectués. Elle est aussi plus coûteuse.

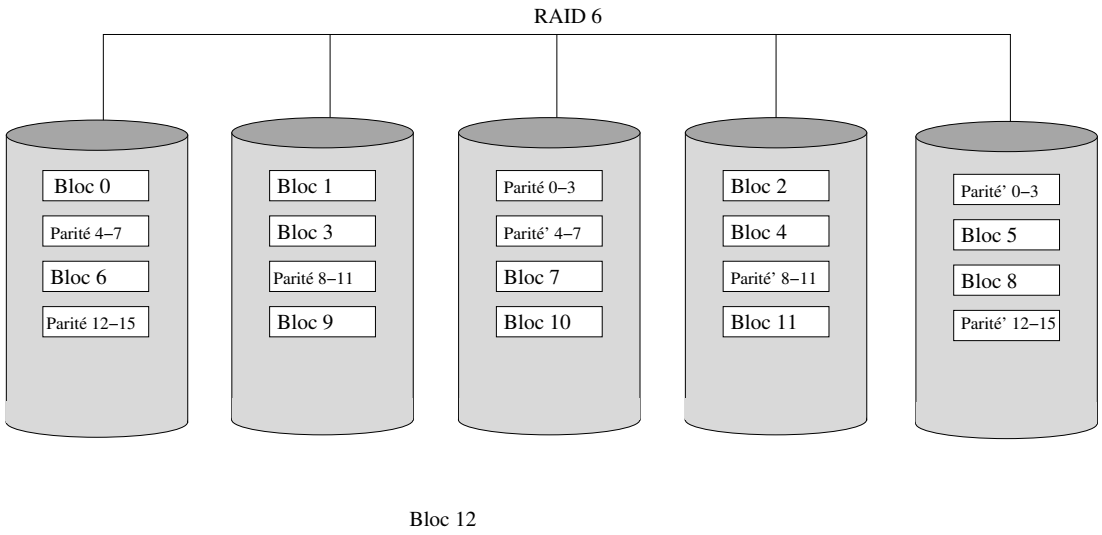

Figure A.8 – Architecture RAID de niveau 6

A.2 Mémoire stable

La mémoire stable [115, 90, 104] est un concept par lequel on tente de garder en permanence la mémoire (disque ou journal) dans un état cohérent. Ainsi, ce qui réside en mémoire stable n'est jamais, par définition, ni perdu ni corrompu même en présence de pannes matérielles ou logicielles.

Pour y parvenir on doit dupliquer l'information et rendre les opérations atomiques. Les données sont copiées sur plusieurs disques afin de tolérer les pannes matérielles reliées à un disque. Les écritures sont rendues atomiques afin de garantir qu'une éventuelle panne, lors d'une mise à jour, ne provoquera aucune corruption.

Pour bien comprendre les problématiques reliées à l'implantation de la mémoire stable, il faut analyser chacun des événements susceptibles de se produire lors d'une opération sur disque. Soit :

1. l'opération se termine avec succès ;

   Dans ce cas, toutes les copies des données sont mises à jour correctement. Les modèles considèrent généralement qu'une copie même correctement écrite peut spontanément se corrompre. Toutefois, ces événements sont si peu fréquents, que l'on considère comme nulle la probabilité que toutes les copies soient affectées simultanément, même s'il n'y en a que deux.

2. une panne pendant l'opération d'écriture ;

   La conséquence d'une panne lors de l'opération d'écriture est une potentielle corruption des données sur une copie ou une mise à jour partielle de seulement un sous-ensemble des copies des données. Toutefois, on considère que les erreurs sont toutes détectables (la probabilité qu'une erreur indétectable se produise est considérée comme négligeable [104]).

3. une panne avant l'opération d'écriture.

   Dans ce cas, la panne s'étant produite avant l'opération, les données antérieures demeurent intactes.

En supposant que l'on conserve deux copies des données, les opérations en mémoire stable sont implantées de la façon suivante :

- Écriture stable ;

  Lors d'une écriture dite stable, on effectue d'abord l'écriture des données sur un premier disque physique (copie primaire) et, seulement lorsque cette opération est terminée avec succès, on reprend l'écriture des données sur le second disque.

  Avant de déclarer une écriture (l'originale ou la copie) terminée avec succès, l'information écrite est relue à des fins de validation. Si cette lecture n'est pas conforme, les opérations d'écriture et de lecture sont ré-exécutées jusqu'à ce qu'elles le deviennent ou aient atteint un certain nombre d'essais maximum. Si cette dernière éventualité se produit, il y aura tentative d'écrire des données à un endroit différent du disque.

  S'il n'y aucune panne, les données sont considérées correctement écrites sur les deux disques.

- Lecture stable ;

  Une lecture stable s'effectue d'abord sur la copie située sur le premier disque. S'il y a une erreur, la lecture est relancée. Si après n essais le résultat est toujours erroné, la lecture est lancée sur la seconde copie. En supposant que l'écriture est toujours correcte et que les deux copies ne peuvent pas se corrompre simultanément, cette seconde lecture terminera avec succès. Par contre, si la corruption simultanée est possible, la création de plus de deux copies doit être envisagée.

- Reprise après une panne.

  Lorsqu'une panne se produit, une opération de récupération est lancée pour vérifier la validité des informations présentes sur les deux disques. Si les deux copies sont identiques, aucune action n'est entreprise. Si l'une des copies est corrompue, une nouvelle copie est créée avec l'information valide disponible sur la seconde copie. Si les deux copies contiennent des informations différentes, la contenu de la première est retranscrit sur la deuxième ou vice versa selon les auteurs [104, 90]).

  Cette façon de faire nous assure qu'une écriture termine avec succès ou qu'aucune modification n'est faite.

En l'absence de panne totale du processeur, cette implantation fonctionne très bien car l'écriture stable produit toujours deux copies valides. Évidemment, cela suppose que notre hypothèse, à savoir que les deux blocs ne peuvent spontanément se corrompre en même temps, tient la route.

Les implantations se généralisent aussi à plus de deux copies de sauvegarde mais, même si cela diminue encore le risque de pertes de données, le fait est que deux copies suffisent généralement pour obtenir un système offrant une très bonne stabilité.

Dans le but d'améliorer la performance de la mémoire stable, des optimisations sont présentées par Tanenbaum[104] et Peterson[90].

A.3 Conclusion

Enfin, il existe d'autres architectures RAID [121, 4, 86, 77, 90], principalement des cas hybrides issus de celles présentées dans ce document.

Notons de plus que les architectures RAID s'implantent aussi bien au niveau logiciel, matériel ou en une combinaison des deux [121, 77, 86].

Bibliographie

[1] Ts. Mohd Hakim Abdul Hamid, J Jais et M Azman : Mosix : Implementation, trend and challenges. 06 2005.

[2] Bit2Me Academy : What is a byzantine failure ? https://academy.bit2me.com/en/what-is-Byzantine-fault/, 2022.

[3] Bit2Me Academy : What is byzantine fault tolerance (bft) ? https://academy.bit2me.com/en/what-is-tolerance-byzantine-failures-bft/, 2022.

[4] Sylvain ADAMI : Le raid et ses différents types. https://www.supinfo.com/articles/single/1176-raid-ses-differents-types, 2015.

[5] Sarita V. Adve et Kourosh Gharachorloo : Shared memory consistency models : A tutorial. *Computer*, 29(12):66–76, dec 1996.

[6] Kshemkalyani Ajay : Chapter 12 : Distributed shared memory. https://www.cs.uic.edu/~ajayk/Chapter12.pdf, 2010.

[7] Hesam Nejati Sharif Aldin, Hossein Deldari, Mohammad Hossein Moattar et Mostafa Razavi Ghods : Consistency models in distributed systems : A survey on definitions, disciplines, challenges and applications. *CoRR*, abs/1902.03305, 2019.

[8] ARMDevelopper : Arm cortex-a series programmer's guide for armv8-a. https://developer.arm.com/documentation/den0024/a/Memory-Ordering, 2012.

[9] Remzi H. Arpaci-Dusseau et Andrea C. Arpaci-Dusseau : Andrew file system. https://pages.cs.wisc.edu/~remzi/OSTEP/dist-afs.pdf, 2018.

[10] Blaise Barney : Introduction to parallel computing. https://computing.llnl.gov/tutorials/parallel_comp/\#MemoryArch, Juin 2018.

[11] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach et Akhilesh Singhania : The multikernel : A new os architecture for scalable multicore systems. *In Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 29–44, New York, NY, USA, 2009. Association for Computing Machinery.

[12] Doron Ben Cohen : What is nfs ? understanding the network file system. https://www.atera.com/blog/what-is-nfs-understanding-the-network-file-system/, 2021.

[13] Roberto Bisiani et Mosur Ravishankar : Plus : a distributed shared-memory system. *In ISCA '90*, 1990.

[14] James Bornholt : Memory consistency models : A tutorial. `https://www.cs.utexas.edu/~bornholt/post/memory-models.html`, 2016.

[15] K. Mani Chandy, Jayadev Misra et Laura M. Haas : Distributed deadlock detection. *ACM Trans. Comput. Syst.*, 1(2):144–156, may 1983.

[16] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley et Edward D. Lazowska : Sharing and protection in a single-address-space operating system. *ACM Trans. Comput. Syst.*, 12(4): 271–307, nov 1994.

[17] David Cheriton : The v distributed system. *Commun. ACM*, 31(3):314–333, mar 1988.

[18] Adaptative Computing : Moab workload manager overview. `http://docs.adaptivecomputing.com/suite/8-0/basic/help.htm#topics/moabWorkloadManager/topics/intro/productOverview.htm`, 2014.

[19] Jonathan Corbet : Coscheduling : simultaneous scheduling in control groups. *LWN (Linux Weekly News)*, septembre 2018.

[20] George Coulouris, Jean Dollimore, Tim Kindberg et Gordon Blair : *Distributed Systems : Concepts and Design*. Addison-Wesley Publishing Company, USA, 5th édition, 2011.

[21] Lyle Daly : What is byzantine fault tolerance ? `https://www.fool.com/investing/stock-market/market-sectors/financials/cryptocurrency-stocks/byzantine-fault-tolerance/`, 2022.

[22] Damien : Consensus series, part 1 : The case for distributed consensus. `https://findora.org/2020/05/the-case-for-distributed-consensus/`, 2021.

[23] Damien : Distributed consensus and the byzantine generals problem : Consensus series, part 2. `https://findora.org/2020/07/distributed-consensus-and-the-byzantine-generals-problem/`, 2021.

[24] Gordon Darroch : Elc : Spacex lessons learned. `https://lwn.net/Articles/540368/`, Mars 2013. [En ligne ; 6-mars-2013].

[25] Neha N. Dawaldi et Manta C. Padole : Comparative study of clock synchronization algorithms in distributed systems. *Advances in Computational Sciences and Technology*, 10(6):1941–1952, 2017.

[26] G. Delp, A. Sethi et D. Farber : An analysis of memnet—an experiment in high-speed shared-memory local networking. *SIGCOMM Comput. Commun. Rev.*, 18(4):165–174, aug 1988.

[27] D.M. Dhamdhere : *Operating Systems : A Concept-based Approach*. Tata McGraw-Hill Pub., 2012.

[28] EasyTechJunky : What is false sharing ? `https://www.easytechjunkie.com/what-is-false-sharing.htm`, 2022.

[29] EatYourBytes : Ddr4 ram data transfers, speed, latency and voltage. https://www.eatyourbytes.com/ddr4-ram-data-transfers-speed-latency-and-voltage/, 2021.

[30] M. Rasit Eskicioglu et T. Anthony Marsland : Distributed Shared Memory : A review. Rapport technique TR96-22, University of Alberta, Department of Computing Science, Edmonton, Alberta, Canada, septembre 1996. https://webdocs.cs.ualberta.ca/~tony/TechnicalReports/TR96-22.pdf.

[31] Expertiza : Memory consistency. https://expertiza.csc.ncsu.edu/index.php/CSC/ECE_506_Spring_2015/7a_ap, 2015.

[32] Ronald Fagin, Joseph Y. Halpern, Yoram Moses et Moshe Y. Vardi : Common knowledge revisited. *Annals of Pure and Applied Logic*, 96(1):89–105, 1999.

[33] Ron Feigen, Alan Skousen et Donald S. Miller : The sombrero distributed single address space operating system. *In OPSR*, 2000.

[34] Mark Fienup : Computer architecture- lecture 28 : Network connected multiprocessors. University of Northern Iowa, Fall 2007.

[35] GeeksForGeeks : Introduction of novell netware. https://www.geeksforgeeks.org/introduction-of-novell-netware/, 2020.

[36] GeeksForGeeks : Andrew file system. https://www.geeksforgeeks.org/andrew-file-system/, 2021.

[37] Ed Gehringer : Csc/ece 501 : Operating system principles - memory consistency. https://people.engr.ncsu.edu/efg/501/f98/lectures/annotations/a20b.html, 1998.

[38] Sanjay Ghemawat, Howard Gobioff et Shun-Tak Leung : The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, oct 2003.

[39] Sanjay Ghemawat, Howard Gobioff et Shun-Tak Leung : The google file system. *In Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, page 29–43, New York, NY, USA, 2003. Association for Computing Machinery.

[40] Dale Grit : Deadlock avoidance,cs 551 : Distributed operating systems. https://www.cs.colostate.edu/~cs551/CourseNotes/Deadlock/DDAvoidance.html, 2003.

[41] Dale Grit : Types of consistency. https://www.cs.colostate.edu/~cs551/CourseNotes/Consistency/TypesConsistency.html, 2003.

[42] Erik Hagersten, Anders Landin et Seif Haridi : Ddm : A cache-only memory architecture. *Computer*, 25(9):44–54, sep 1992.

[43] Joseph Y. Halpern et Yoram Moses : Knowledge and common knowledge in a distributed environment. *J. ACM*, 37(3):549–587, jul 1990.

[44] Gernot Heiser, Kevin Elphinstone, Jerry Vochteloo, Stephen Russell et Jochen Liedtke : The mungi single-address-space operating system. *Software : Practice and Experience*, 28, 1998.

[45] Thomas A. Henzinger et Christoph M. Kirsch, éditeurs. *Embedded Software, First International Workshop, EMSOFT 2001, Tahoe City, CA, USA, October, 8-10, 2001, Proceedings*, volume 2211 de *Lecture Notes in Computer Science*. Springer, 2001.

[46] Liviu Iftode, Jaswinder Pal Singh et Kai Li : Scope consistency : A bridge between release consistency and entry consistency. *In* Anon, éditeur : *Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 277–287, décembre 1996. Proceedings of the 1996 8th Annual ACM Symposium on Parallel Algorithms and Architectures ; Conference date : 24-06-1996 Through 26-06-1996.

[47] Living in Belgium : Différence entre uma et numa. `https://fr.living-in-belgium.com/difference-between-uma-and-numa-44`, 2021.

[48] Culture Informatique : C'est quoi le raid ? `https://www.culture-informatique.net/cest-quoi-raid`, 2017.

[49] InformIT : Multiprocessor operating systems. `https://www.informit.com/articles/article.aspx?p=26027#`, 2002.

[50] Jetbrains : Is compile time reordering of instructions better than run time reordering ? `https://www.quora.com/Is-compile-time-reordering-of-instructions-better-than-run-time-reordering`, 2018.

[51] Kerrighed : Kerrighed : Linux clusters made easy. `http://www.kerrighed.org/wiki/index.php/Main_Page`, 2010.

[52] Ajay D. Kshemkalyani et Mukesh Singhal : *Distributed Computing : Principles, Algorithms, and Systems*. Cambridge University Press, USA, 1 édition, 2008. Voir chapitre 17.

[53] Leslie Lamport : Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[54] Leslie Lamport : How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers C-28*, 9:690–691, September 1979.

[55] Leslie Lamport, Robert E. Shostak et Marshall C. Pease : The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[56] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz et M.S. Lam : The stanford dash multiprocessor. *Computer*, 25(3):63–79, 1992.

[57] Kai Li et Paul Hudak : Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst.*, 7(4):321–359, nov 1989.

[58] Tom Lianza et Snook Chris : A byzantine failure in the real world. `https://blog.cloudflare.com/a-byzantine-failure-in-the-real-world/`, 2020.

[59] LinuxPMI : Linuxpmi. `http://linuxpmi.org/trac/`, 2021.

[60] Hari Haranath Madduri : *Deadlock Avoidance in Distributed Operating Systems (Resource Allocation, Starvation, Banker's Algorithm)*. Thèse de doctorat, 1985. AAI8528433.

[61] Mamoru Maekawa : A square root n algorithm for mutual exclusion in decentralized systems. *ACM Trans. Comput. Syst.*, 3(2):145–159, mai 1985.

[62] Musing Mortoray : Cpu reordering – what is actually being reordered ? `https://mortoray.com/2010/11/18/cpu-reordering-what-is-actually-being-reordered/`, 2010.

[63] David Mosberger : Memory consistency models. *SIGOPS Operating Systems Review*, 27(1):18–26, jan. 1993.

[64] MOSIX : Mosix : Cluster management system. `https://mosix.cs.huji.ac.il/txt_about.html`, 2021.

[65] Alex Moskov : What is the byzantine generals problem ? `https://coincentral.com/byzantine-generals-problem/`, 2018.

[66] Sape J. Mullender, Guido van Rossum, Andrew S. Tanenbaum, Robbert van Renesse et Hans van Staveren : Amoeba : a distributed operating system for the 1990s. *Computer (New York)*, 23(5):44–53, may 1990.

[67] Henk L. Muller, Paul W. A. Stallard et David H. D. Warren : The data diffusion machine with a scalable point-to-point network. Rapport technique, GBR, 1993.

[68] Vijay Nagarajan, Daniel J. Sorin, Mark D. Hill, David A. Wood et Natalie Enright Jerger : *A Primer on Memory Consistency and Cache Coherence*. Morgan & Claypool Publishers, 2nd édition, 2020.

[69] Preshing on Programming : Memory ordering at compile time. `https://preshing.com/20120625/memory-ordering-at-compile-time/`, 2012.

[70] OpenAFS : Welcome to the home of openafs. `https://www.openafs.org/`, 2021.

[71] OpenPBS : Openpbs. `https://www.openpbs.org/`, 2021.

[72] Openstack : Manila. `https://wiki.openstack.org/wiki/Manila`, 2021.

[73] Openstack : Swift. `https://wiki.openstack.org/wiki/Swift`, 2021.

[74] Openstack : Welcome to swift's documentation ! `https://docs.openstack.org/swift/latest/`, 2021.

[75] John K. Ousterhout : Scheduling techniques for concurrent systems. *In Proceedings of the 3rd International Conference on Distributed Computing Systems (ICDCS)*, pages 22–30, 1982.

[76] Benjamin Pierce : Mobile agent computing : A white paper. `https://www.cis.upenn.edu/~bcpierce/courses/629/papers/Concordia-WhitePaper.html`, 2002.

[77] Jean-François Pilou, Galilée et et al. : Le raid c'est quoi ? `https://www.commentcamarche.net/faq/159-le-raid-c-est-quoi`, 2016.

[78] Jelica Protic, Milo Tomasevic et Veljko Milutinovic : A survey of distributed shared memory systems. pages 74–84 vol.1, 02 1995.

[79] Ques10 : Short note on deadlock avoidance algorithm in distributed systems. https://www.ques10.com/p/2206/short-note-on-deadlock-avoidance-algorithm-in-dist/?, 2020.

[80] RedHat : Chapter 18. network file system (nfs). https://web.mit.edu/rhel-doc/5/RHEL-5-manual/Deployment_Guide-en-US/ch-nfs.html, 2021.

[81] Glenn Ricart et Ashok K. Agrawala : An optimal algorithm for mutual exclusion in computer networks. *Commun. ACM*, 24(1):9–17, jan 1981.

[82] Sami Rollins : Cs686 - time synchronization. https://www.cs.usfca.edu/~srollins/courses/cs686-f08/web/notes/timesync.html, 2008.

[83] César Sánchez, Henny B. Sipma et Zohar Manna : A family of distributed deadlock avoidance protocols and their reachable state spaces. *In* Matthew B. Dwyer et Antónia Lopes, éditeurs : *Fundamental Approaches to Software Engineering*, pages 155–169, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[84] Ichiro Satoh : Mobile agents. https://research.nii.ac.jp/~ichiro/papers/satoh-mobile-agent.pdf, 2008.

[85] Jessica Scarpati : Network time protocol (ntp). https://www.techtarget.com/searchnetworking/definition/Network-Time-Protocol, 2022.

[86] Amen School : Le raid c'est quoi ? https://www.amenschool.fr/raid-informatique-quest-ce-que-cest/, 2017.

[87] SemanticScholar : Single address space operating system. https://www.semanticscholar.org/topic/Single-address-space-operating-system/465770, 2022.

[88] SchedMD : Slurm workload manager. https://slurm.schedmd.com/documentation.html, 2021.

[89] Abraham Silberschatz, Peter B. Galvin et Greg Gagne : *Operating System Concepts*. Wiley Publishing, 9th édition, 2012.

[90] Abraham Silberschatz, Peter B. Galvin et Greg Gagne : *Operating System Concepts*. Wiley Publishing, 9th édition, 2012.

[91] Abraham Silberschatz, Peter Baer Galvin et Greg Gagne : *Operating System Concepts, 10th Edition*. Wiley, 2018.

[92] Rick Simpson : Just how large is a 64-bit address ? https://cnx.org/contents/H2P-PZyF@2/Just-how-large-is-a-64-bit-address, 2022.

[93] M. Singhal et N.G. Shivaratri : *Advanced Concepts in Operating Systems : Distributed, Database, and Multiprocessor Operating Systems*. Computer Science Series. McGraw-Hill, 1994.

[94] Kevin Sookocheff : How does ntp work ? https://sookocheff.com/post/time/how-does-ntp-work/, 2021.

[95] Daniel Sorin, Mark Hill et David Wood : *A Primer on Memory Consistency and Cache Coherence*, volume 6. 11 2011.

[96] CORPORATE SPARC International, Inc. : *The SPARC Architecture Manual (Version 9)*. Prentice-Hall, Inc., USA, 1994.

[97] Inc. Staff Sparc International et SPARC International : *The SPARC Architecture Manual : Version 8*. SPARC International. Prentice Hall, 1992.

[98] StackOverflow : Gcc's reordering of read/write instructions. `https://stackoverflow.com/questions/22106843/gccs-reordering-of-read-write-instructions`, 2014.

[99] StackOverflow : How does cpu reorder instructions. `https://stackoverflow.com/questions/39062100/how-does-cpu-reorder-instructions`, 2016.

[100] William Stallings : *Operating Systems : Internals and Design Principles*. Prentice Hall Press, USA, 7th édition, 2011.

[101] Anthony Stevens : Understanding the byzantine generals' problem (and how it affects you). `https://medium.com/coinmonks/a-note-from-anthony-if-you-havent-already-please-read-the-article-gaining-clarity-on-key-787989107969`, 2018.

[102] M.C. Tam et D. Farber : Capnet-an approach to ultra high speed network. *In IEEE International Conference on Communications, Including Supercomm Technical Sessions*, volume 3, pages 955–961, 1990.

[103] Andrew S. Tanenbaum : *Distributed operating systems*. Prentice Hall, 1995.

[104] Andrew S. Tanenbaum et Herbert Bos : *Modern Operating Systems*. Prentice Hall Press, USA, 4th édition, 2014.

[105] Andrew S. Tanenbaum, Robbert van Renesse, Hans van Staveren, Gregory J. Sharp et Sape J. Mullender : Experiences with the amoeba distributed operating system. *Commun. ACM*, 33(12):46–63, dec 1990.

[106] Andrew S. Tanenbaum et Maarten van Steen : *Distributed Systems : Principles and Paradigms*. Pearson Prentice Hall, Upper Saddle River, NJ, 2 édition, 2007.

[107] A.S. Tanenbaum et M. van Steen : *Distributed Systems*. CreateSpace Independent Publishing Platform, 2017.

[108] TechDifference : Difference between uma and numa. `https://techdifferences.com/difference-between-uma-and-numa.html`, 2021.

[109] Colorado State University : Cs 551 : Distributed operating systems - notes on consistency & replication. `https://www.cs.colostate.edu/~cs551/CourseNotes/Consistency/ConsReplTOC.html`, 2003.

[110] Wiki : Single address space operating system. `https://wiki.c2.com/?SingleAddressSpaceOperatingSystem`, 2022.

[111] Wikipedia : Stanford dash. https://en.wikipedia.org/wiki/Data_diffusion_machine, 2015.

[112] Wikipedia : Andrew file system. https://fr.wikipedia.org/wiki/Andrew_File_System, 2018.

[113] Wikipedia : Mosix. https://fr.wikipedia.org/wiki/MOSIX, 2018.

[114] Wikipedia : Shared memory. https://en.wikipedia.org/wiki/Shared_memory, Octobre 2018.

[115] Wikipedia : Stable storage. https://en.wikipedia.org/wiki/Stable_storage, 2018.

[116] Wikipedia : Uniform memory access. https://en.wikipedia.org/wiki/Uniform_memory_access, Décembre 2018.

[117] Wikipedia : Google file system. https://fr.wikipedia.org/wiki/Google_File_System, 2019.

[118] Wikipedia : Slurm. https://fr.wikipedia.org/wiki/SLURM, 2019.

[119] Wikipedia : Kerrighed. https://fr.wikipedia.org/wiki/Kerrighed, 2020.

[120] Wikipedia : Processor consistency. https://en.wikipedia.org/wiki/Processor_consistency, Avril 2020.

[121] Wikipedia : Raid (informatique). https://fr.wikipedia.org/wiki/RAID_(informatique), 2020.

[122] Wikipedia : Amoeba (operating system). https://en.wikipedia.org/wiki/Network_File_System, 2021.

[123] Wikipedia : Andrew file system. https://en.wikipedia.org/wiki/Andrew_File_System, 2021.

[124] Wikipedia : Andrew file system. https://en.wikipedia.org/wiki/OpenAFS, 2021.

[125] Wikipedia : Arrivé-avant. https://fr.wikipedia.org/wiki/Arriv%C3%A9-avant, 2021.

[126] Wikipedia : Atomic broadcast. https://en.wikipedia.org/wiki/Atomic_broadcast, 2021.

[127] Wikipedia : Coscheduling. https://en.wikipedia.org/wiki/Coscheduling, 2021.

[128] Wikipedia : Distributed shared memory. https://en.wikipedia.org/wiki/Distributed_shared_memory, 2021.

[129] Wikipedia : False sharing. https://en.wikipedia.org/wiki/False_sharing, 2021.

[130] Wikipedia : Google file system. https://en.wikipedia.org/wiki/Google_File_System, 2021.

[131] Wikipedia : Idempotence. https://fr.wikipedia.org/wiki/Idempotence, 2021.

[132] Wikipedia : Linearizability. https://en.wikipedia.org/wiki/Linearizability, 2021.

[133] Wikipedia : Mach (kernel). https://en.wikipedia.org/wiki/Mach_(kernel), 2021.

[134] Wikipedia : Maui cluster scheduler. https://en.wikipedia.org/wiki/Maui_Cluster_Scheduler, 2021.

[135] Wikipedia : Memory ordering. https://en.wikipedia.org/wiki/Memory_ordering, 2021.

[136] Wikipedia : Mobiles agent. https://en.wikipedia.org/wiki/Mobile_agent, 2021.

[137] Wikipedia : Mosix. https://en.wikipedia.org/wiki/MOSIX, 2021.

[138] Wikipedia : Netware core protocol. https://en.wikipedia.org/wiki/NetWare_Core_Protocol, 2021.

[139] Wikipedia : Network file system. https://fr.wikipedia.org/wiki/Network_File_System, 2021.

[140] Wikipedia : Network file system. https://en.wikipedia.org/wiki/Network_File_System, 2021.

[141] Wikipedia : openmosix. https://en.wikipedia.org/wiki/OpenMosix, 2021.

[142] Wikipedia : Panne byzantine. https://fr.wikipedia.org/wiki/Panne_byzantine, 2021.

[143] Wikipedia : Portable batch system. https://en.wikipedia.org/wiki/Portable_Batch_System, 2021.

[144] Wikipedia : Serializability. https://en.wikipedia.org/wiki/Serializability, 2021.

[145] Wikipedia : Slurm. https://en.wikipedia.org/wiki/Slurm_Workload_Manager, 2021.

[146] Wikipedia : Tomasulo algorithm. https://en.wikipedia.org/wiki/Tomasulo_algorithm, 2021.

[147] Wikipedia : Torque. https://en.wikipedia.org/wiki/TORQUE, 2021.

[148] Wikipedia : V (operating system). https://en.wikipedia.org/wiki/V_(operating_system), 2021.

[149] Wikipedia : Byzantine fault. https://en.wikipedia.org/wiki/Byzantine_fault, 2022.

[150] Wikipedia : Cohérence (données). https://fr.wikipedia.org/wiki/CohÃĭrence_(donnÃĭes), 2022.

[151] Wikipedia : Consensus (computer science). https://en.wikipedia.org/wiki/Consensus_(computer_science), 2022.

[152] Wikipedia : Consistency model. https://en.wikipedia.org/wiki/Consistency_model, 2022.

[153] Wikipedia : Eventual consistency. https://en.wikipedia.org/wiki/Eventual_consistency, 2022.

[154] Wikipedia : Exécution dans le désordre. https://fr.wikipedia.org/wiki/ExÃľcution_dans_le_dÃľsordre, 2022.

[155] Wikipedia : Ibm system/370. https://en.wikipedia.org/wiki/IBM_System/370, 2022.

[156] Wikipedia : Lamport timestamp. https://en.wikipedia.org/wiki/Lamport_timestamp, 2022.

[157] Wikipedia : Network time protocol. https://fr.wikipedia.org/wiki/Network_Time_Protocol, 2022.

[158] Wikipedia : Single address space operating system. https://en.wikipedia.org/wiki/Single_address_space_operating_system, 2022.

[159] Wikipedia : Vxworks. https://en.wikipedia.org/wiki/VxWorks, 2022.

[160] Simple English Wikipedia : Deadlock. https://simple.wikipedia.org/wiki/Deadlock, 2021.

[161] Tim Wilkinson, Kevin Murray, Stephen Russell, Gernot Heiser et Jochen Liedtke : Single address space operating systems. 1990.

[162] A. Wilson, M. Teller, T. Probert, Dyung Le et R. LaRowe : Lynx/galactica net : a distributed, cache coherent multiprocessing system. *Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences*, i:416–426 vol.1, 1992.

[163] Y.C. Yeh : Safety critical avionics for the 777 primary flight controls system. *In 20th DASC. 20th Digital Avionics Systems Conference (Cat. No.01CH37219)*, volume 1, pages 1C2/1–1C2/11, 2001.

[164] YourDictionary : Simple catch-22 examples. https://examples.yourdictionary.com/simple-catch-22-examples.html, 2022.