Systèmes et Réseaux
Support de Cours pour le M1 EIDD

Nicolas Blanchard

Premier semestre 2016

Abstract

Ce poly sommaire n'est pas un cours complet, mais un ensemble de pointeurs et un résumé des concepts expliqués en cours (je vous conseille donc de faire des recherches sur les sujets abordés que vous n'avez pas bien compris). Son but est de rappeler quels concepts ont été abordés (pour l'examen), de rappler des concepts généraux et d'apporter une vision d'ensemble de la programmation système et réseau. Pour un livre détaillé approfondissant tout cela je recommande le livre "UNIX : Programmation et communication" de J.-M. Rifflet et J.-B. Yunes. Pour celles ayant de l'expérience et désirant juste un rappel des différentes fonctions je conseille aussi ce guide.

Contents

1 Architecture des ordinateurs et Hardware

1.1 Histoire

Depuis les travaux fondateurs de Turing et Von Neumann, les outils pratiques comme théorique ont fortement changé. Comme prévu par la loi de Moore[1], la taille des composants électroniques a chuté de manière exponentielle, mais les avancées pratiques en architecture ont aussi fortement changé la donne. Avec l'arrivée du transistor, puis la démocratisation des PC, celle d'internet, des smartphones et bientôt de l'internet des objets, nous vivons dans une révolution technologique permanente, sans indication de ralentissement. Ainsi, il est désormais impossible de suivre en détail la progression de l'exécution d'un programme dans une machine, et le code est de plus en plus produit par des machines elles-mêmes (par l'autogénération grâce à des langages de très haut niveau).

1.2 Transistors

Développé en 1947, le transistor permet grâce à des semi-conducteurs d'amplifier un signal sans avoir de parties mobiles (contrairement aux amplis à lampe). Avec quelques transistors, on peut aussi former des portes logiques (dont la porte NAND qui en nécessite seulement 2, à la place de 4 pour un AND ou OR). On peut donc avoir des systèmes purement électroniques faisant du calcul. Cela permet non seulement de miniaturiser assez facilement le calculateur mais aussi d'atteindre des hautes fréquences. Grâce à cette miniaturisation, nos circuits imprimés sont désormais gravés à quelques nanomètres près, et on se rapproche de la frontière de l'atome. Cependant, on est aujourd'hui très proche de la limite car les courants parasites deviennent ingérables et les effets quantiques commencent à apparaître (sans parler des problèmes de refroidissement).

1.3 Composants

Chaque composant d'un ordinateur est donc fabriqué à partir de transistors, pouvant atteindre plusieurs milliards pour un processeur. Si au départ tout le calcul était géré par ce dernier (mais ni mémoire, ni I/O), la spécialisation progressive a changé cela. Un processeur reste un outil de calul SISD : une instruction une donnée. Pour des tâches répétitives comme le calcul vectoriel, on préfère avoir des calculateurs parallèles pouvant faire efficacement du SIMD (single instruction multiple data), ce qui est géré par nos cartes graphiques. Par contre, vu que les contraintes d'accès mémoire causent des délais importants, une partie de celle-ci est désormais directement gérée à l'intérieur du processeur. De même, les cartes son sont spécialisées en traitement de signal analogique, et les cartes réseaux peuvent parfois gérer des signaux à très haute fréquence. Cette spécialisation et les avancées théorique sont responsables de près d'un tiers des augmentations de performance annuelles, mais complexifient grandement le travail du développeur.

[1] D'autres variantes de cette loi sont aussi importantes, comme le fait que le coût les usines de fabrication de semi-conducteurs suit lui aussi une croissance exponentielle.

1.4 Ce qui se passe au démarrage

Quand on appuie sur l'interrupteur d'un ordinateur, que se passe-t-il donc ? Tout d'abord, avant d'être allumé, un système minimal est tout de même activé, marchant sur du 5V. Ce système alimente continûment une sous-partie de la carte-mère et permet d'avoir un démarrage programmé à une certaine heure, ou lorsqu'un signal arrive par le réseau, et permet aussi de conserver un peu d'énergie pour maintenir l'horloge. C'est la raison pour laquelle il faut manuellement appuyer sur le bouton de l'alimenation (située à l'arrière du boîtier) lorsqu'on démonte un PC, pour ne pas risquer de se faire électrocuter[2]. Lorsque quelqu'un appuie sur l'interrupteur, le processeur charge l'instruction de démarrage, chargeant le BIOS, qui lance certains hardware. Il exécute alors le POST (Power-On Selft Test), responsable d'un bip lorsqu'il y a un problème. Après ce test il détecte le support de stockage principal et regarde si un système d'exploitation (OS) est installé dessus et essaye de booter cet OS. Enfin l'OS lance le premier processus, puis tous les daemons et autres services, avant de donner la main à l'utilisateur qui peut alors lancer ses propres programmes.

1.5 Système d'exploitation

Le but de l'OS est multiple. Il doit gérer le hardware et les drivers, attribuer des ressources (temps cpu et mémoire, via un scheduler) aux processus, et le plus souvent donner une interface à l'utilisateur. Différents types d'OS existent, les plus connus étant Mac et Linux, sur une base UNIX, et Windows, mais aussi les OS mobiles comme Android ou iOS. Un OS a généralement un noyau, et les processus peuvent sont ainsi gérés de manière indépendante et sécurisée. Quand un processus essaye de manipuler ses propres variables il n'y a aucun problème, mais quand il demande des accès matériels (plus de mémoire, un accès à une autre zone mémoire, ou bien à une interface clavier), l'appel passe avant par le noyau qui peut donc abstraire les contraintes matérielles. L'utilisateur a un accès au système via ou bien une interface graphique ou bien une console, mais même pour le superutilisateur (root), seule une partie des fonctionnalités est accessible directement. On peut aussi faire tourner un OS dans une machine virtuelle à l'intérieur d'un autre OS, et ce processus peut désormais être fait avec une efficacité remarquable. Nous nous concentrerons cependant dans ce cours sur le système UNIX.

[2] Éléctriser pour les pédants.

2 Systèmes de fichiers

Quand l'OS veut manipuler et stocker des données, il ne le fait pas de manière directe en écrivant des 0 ou 1 sur son support de stockage, mais de manière structurée. De nombreuses architectures existent et seront discutées après quelques considérations matérielles.

2.1 Fonctionnement mécanique d'un disque dur

Un disque dur est en réalité composé de plusieurs disques (ou plateaux), sur lesquels les données sont encodées de manière magnétique, et d'un ensemble de têtes de lecture. Une seule tête est utilisée à chaque instant, et est positionnée à une certaine distance du centre du plateau qui tourne à haute vitesse. La tête de lecture analyse l'orientation du champ magnétique juste en dessous d'elle et transforme cela en suite de bits. Au départ chaque bit correspondait à une position indiquée par (plateau/rayon/secteur), où un secteur correspond aux données sur un même rayon entre deux angles donnés. Cela donne cependant des grands écarts entre les densité d'information entre le centre du plateau et le bord, donc les techniques sont aujourd'hui un peu plus compliquées (le nombre de secteur augmentant avec le rayon). La notion de secteur a toutefois un autre avantage : si on veut accéder aux données sur un disque, ce n'est pas efficace de faire correspondre à chaque bit (ou à chaque octet) une adresse. En effet, cela donnerait des limites de 4Gb (ou 4Go) pour les systèmes 32bits. À la place on a décidé de faire correspondre un secteur à chaque adresse, donc un espace allant de 512b (historiquement) à 4Ko (Advanced Format, plus récent), et cela permet d'avoir des disques allant jusqu'à 16To en adressage 32bits.

Cette division en secteur et la nécessité de bouger physiquement la tête de lecture pour accéder à un autre secteur (ou rayon) fait que les lectures séquentielles sont beaucoup plus rapides que les lectures aléatoires, les lectures étant séquentielles si elles correspondent à des secteurs voisins. Les systèmes SSD n'ont pas cette contrainte physique et sont plus proches de la RAM (où chaque case a une même latence), mais ont tout de même une différence. Pour comparaison, en lecture ou écriture séquentielle il est normal d'avoir du 150Mo/s pour des disques durs, alors que des lectures sur des secteurs aléatoires fait chuter ces taux d'un facteur 150. Pour les SSD on est plutôt au facteur 10-15 en lecture et 4 en écriture. Il y a une corrélation inverse entre les densités d'informations et la vitesse de lecture (et, de pair, le côté séquentiel de la lecture). Les bandes magnétiques sont ainsi beaucoup plus denses, mais l'accès se fait de manière linéaire (et le stockage par ADN atteint des densités immenses mais la récupération des données est très couteuse).

2.2 Structure du disque

Un disque est généralement séparés en plusieurs partitions. Pour différencier ces partitions, un secteur du disque est réservé pour indiquer ces informations. Deux systèmes coexistent, le GPT (pour GUID Partition Table) et le MBR, plus ancien (et en lente disparition à cause de limites sur la taille des partitions et de difficulté à faire évoluer ce standard). Ces systèmes, écrits dans les premiers (et parfois derniers) secteurs du disque, sont appelés au démarrage pour savoir où sont les différents systèmes d'exploitation qui vont prendre la relève. Ils indiquent les secteurs où les partitions commencent et terminent, si elles sont bootables, et quelques autres flags et informations. Un gestionnaire comme GRUB permet au démarrage de choisir quelle partition on veut charger (GRUB lui-même se bootstrapant sur les premiers secteurs du disque). Certaines partitions ont un rôle particulier : le SWAP qui permet sous Unix de supplémenter la RAM, ou le Windows Recovery Environment qui contient des outils de diagnostics sous windows quand la partition principale ne fonctionne plus.

2.3 Système de fichier

Le système de fichier est au coeur de l'OS, et permet d'avoir une structure permettant d'accéder de manière structurée aux données. Il existe de très nombreux types de systèmes de fichier (Wikipedia en liste entre 100 et 200, sans compter les extensions), ayant des propriétés différentes, mais un certain nombre de points communs. Tout d'abord, ils ont généralement une structure arborescente avec une racine ('/' sous Unix), des dossiers, et dans ces dossiers des fichiers. Ces dossiers et fichiers ont des noms, et une certaine quantité de metadata associée (comme des propriétaires, une taille). Selon le système, on peut avoir une zone réservée pour l'arborescence (ce qui implique un nombre maximal de fichiers réduit), ou alors stocker l'arborescence au même endroit que les données. Généralement, le metadata grandit de manière logarithmique avec la taille des données, et les systèmes de fichier récents utilisent cela, en étant beaucoup plus riches que les systèmes anciens. Stocker ce metadata a cependant un coût : entre 0.2% et 3.5% de l'espace total en général sur des systèmes modernes (mais cela peut dépasser 10% pour certains systèmes de fichier).

Au départ, lorsque l'on crée une partition et que l'on installe un système dessus, les données sont dans des emplacements contigus, mais au fur et à mesure que l'on supprime et crée de nouveaux fichiers, il peut y avoir des "trous", c'est-à-dire des zones libres de petite taille, ou bien des fichiers qui sont obligés d'être coupés en plusieurs parties. Ce processus, la fragmentation, est plus ou moins bien géré par différents systèmes, et nécessite parfois de réécrire de nombreux fichiers de manière optimisée pour augmenter l'espace libre et la vitesse d'accès. Enfin, certains systèmes peuvent changer la taille de leur partition (comme ext4, HFS+ et NTFS), et peuvent ainsi augmenter (ou diminuer) l'espace disponible. Un système notable est le FAT32, qui est très rudimentaire et a de nombreuses limites (partition de taille 2To, fichiers de taille 4Go, profondeur d'arborescence limitée), mais pour ces raisons est bien implémenté sur tous les OS, et sert souvent pour échanger des fichiers (grâce aux clés USB, généralement formattées en FAT32).

2.4 Inode

Sous Unix, tout est fichier, et les fichiers sont indiqués par leurs inodes. Ces derniers contiennent de nombreuses informations dont : la taille du fichier, le volume sur lequel il est stocké, le propriétaire (et groupe), un ensemble de permissions, des filestamps pour les derniers accès (en lecture / écriture) et surtout des pointeurs vers les blocs du disque qui contiennent les données associées au fichier. Cela a de nombreux avantages : bouger un fichier est facile, car il suffit de créer un nouvel inode au bon endroit dans l'arborescence et de le faire pointer vers les mêmes données. On peut aussi avoir plusieurs noms (et chemins) correspondant au même fichier, car il suffit de pointer vers le même inode (mais cela veut dire qu'on ne peut généralement pas trouver le chemin d'un fichier ouvert, vu qu'on a un numéro d'inode, et que cela correspond potentiellement à plusieurs noms et chemins).

Gérer les dossiers est très facile : on a le metadata sur le dossier, et le bloc de données est rempli de noms et de numéros d'inodes correspondant aux descendants. De plus, les fichiers de grandes tailles sont traités de manière logique : l'inode contient quinze pointeurs directs et indirects. Les douze premiers pointent chacun vers un bloc du disque (donc entre 512o et 4Ko), le suivant vers un bloc indirect de niveau 1 (rempli d'adresses de blocs de data), un bloc de niveau 2 (rempli d'adresses de blocs de niveau 1) et un bloc de niveau 3. On a donc des tailles de fichier de 48 Ko, 6Mo, 768Mo et 98Go de manière native avec des blocs de taille 512o (on peut aussi étendre ces limites dans certains cas). Il y a cependant plusieurs contraintes dont on doit se rappeler : tout d'abord il y a un nombre maximal d'inodes qui est fixé à la création de la partition et qui ne peut pas être changé (aux alentours de 4 milliards sur les systèmes 32bits).

Pour en savoir plus sur un inode sous Unix on peut faire appel à ls et stat.

3 Processus

Il y a quelques décénnies, quand on démarrait un ordinateur, un unique programme était lancé, tournait, faisait ses sorties et la machine s'arrêtait (en fait elle écrivait qu'il était désormais possible de couper le courant, avant les soft-switch que l'on a désormais sur nos machines). Ce système étant très limité, il céda la place aux systèmes multi-processus. Plusieurs programmes se partagent donc la mémoire et les capacités de calcul de la machine. Cela est naturellement encore plus difficile lorsqu'on a plusieurs unités de calculs (coeurs) et différentes zones mémoires. Regardons d'abord ce qu'est un processus.

3.1 Processus et programme

Un processus est composé de plusieurs éléments : déjà, le code du programme associé (en langage machine naturellement), ensuite, les variables qu'il utilise, et une zone de mémoire disponible au cas où (ces variables temporaires sont appelées le tas), et les instructions en attente (notamment les appels récursifs, pour savoir où nous en sommes dans le programme). Cela occupe une zone mémoire mais un processus a aussi d'autres caractéristiques, pour pouvoir être géré par l'OS (et qu'on peut voir dans la table des processus). En premier lieu, son statut (actif, en attente, en marche ou zombie), son PID, un numéro d'identité propre au processus et attribué par le noyau (on peut connaître son pid en faisant getpid()). Ensuite, son propriétaire, pour savoir les permissions du processus, et sa priorité (important pour le scheduler). Le processus peut aussi être en background ou foreground (selon qu'il rende la main à l'utilisateur dans la console pendant son exécution ou pas). Il dispose enfin de son PPID, c'est-à-dire le PID de son processus père (qu'on peut avoir en faisant getppid()). En effet, chaque processus possède un père (et parfois des fils). C'est dû au fait que lors du démarrage, un unique processus est lancé et va créer de nombreux enfants qui vont eux-mêmes créer d'autres processus.

3.2 Forks

Ce premier processus va donc créer de nombreux descendants et pour faire cela on utilise la primitive fork(). Cette dernière va créer une copie du processus en cours, en copiant toute la mémoire, les instructions en cours et le reste. On va donc se retrouver avec deux processus qui sont identiques excepté pour une variable : la valeur de retour du fork(). Pour l'un des deux (qu'on appelle le fils), cette variable sera égale à 0, et pour le père elle sera égale au PID du fils. Cela permet de différencier facilement les deux et de leur faire executer des codes différents par la suite, selon la valeur de retour du fork. Quelques remarques sur le fork :

- Fork() duplique les variables et leurs valeurs. Toute variable qui n'est pas modifiée dans la suite du code sera donc égale dans les deux processus.
- Ces variables ne sont cependant pas partagées, donc si le père (ou le fils) modifie une variable, la valeur de la variable correspondante dans l'autre processus ne sera pas affectée. Pour faire cela il faut faire de la communication entre processus (chapitre 5).
- De même, deux processus qui viennent de fork ne vont à priori pas s'exécuter de manière synchrone (un peut avoir 500 cycles d'horloge pendant que l'autre en a 10), avoir une syncrhonisation est donc plus difficile et dépend du scheduler.
- Si on fait k forks d'affilée, on ne se retrouve pas avec $k+1$ processus mais bien avec 2^k, car à chaque nouveau fork() chaque processus doit l'exécuter et le nombre double donc.
- Un processus père a le PID du fils et peut donc lui envoyer des signaux (chapitre 5), mais le fils peut aussi obtenir celui du père avec getppid().

- Un processus père est généralement supposé attendre (wait) la mort de ses fils avant de terminer. En effet, quand un processus termine, il envoie un signal à son père reste encore en mémoire jusqu'à ce que son père s'en rende compte (ce qu'on peut faire avec un handler). Il est alors dans un état zombie. Si le père termine avant (ou sans) faire cela, les processus récupèrent un autre père (qui n'est pas son grand-père mais le processus init, de PID 1).

3.3 Ordonnancement (scheduling)

Comme nous avons désormais de nombreux processus sur la même machine, il est nécessaire de gérer leur exécution pour qu'aucun ne prenne toutes les ressources. C'est un des rôles principaux de l'OS, qui utilise généralement des variantes du round-robin. Cette technique, où chacun fait un nombre constant d'opérations à la suite avant de passer la main, garantit un certain équilibre entre les différentes tâches. Elle a cependant d'autres problèmes : déjà, changer de processus a un coût, car il faut recharger toutes les variables nécessaires en mémoire, ce qui prend du temps, et les processus à haute priorité peuvent se retrouver sans ressource à un moment critique. Changer trop fréquemment va augmenter l'overhead, et le contraire peut causer des bugs dans les systèmes critiques. C'est le rôles du scheduler (qui a des algorithmes souvent assez complexes) de gérer cela bien, et les processus eux-mêmes n'ont presque pas de contrôle sur leur exécution. La seule option pour le processus est le niceness, qui vaut entre -20 et 19 (mais où seul le root peut imposer des valeurs négatives). Cette niceness est rajoutée à la priorité du processus (plus la prioriété est faible et plus le processus a le droit aux ressources). Cela permet donc de manuellement diminuer la quanité de ressources disponibles pour un processus (sauf pour le root). Les techniques utilisées sont d'autant plus complexes que les programmes nécessitent souvent des ressources (comme le disque dur), et l'optimisation est difficile et se fait généralement à base d'heuristiques. Quelques méthodes classiques de scheduling toutefois : le FIFO, où le premier processus est servi jusqu'à terminaison (ce n'est pas raisonnable sur nos machines, mais faisable dans certaines conditions où on a des garanties de terminaisons). On peut aussi prendre le processus de plus petite priorité, ou bien celui minimisant le temps avant terminaison (qui ne peut souvent être qu'estimé). Si le système est préemptif, ce qui veut dire qu'on peut interrompre un processus au milieu de son exécution pour le reprendre plus tard, on peut utiliser des round-robins (potentiellement pondérés par les priorités). Le champ de l'ordonnancement est très riche et a de nombreuses applications dans l'industrie (notamment pour la gestion d'usine).

4 Gestion mémoire

Tous ces processus ont besoin de stocker leurs variables et leur code quelque part. On pourrait bien sûr tout stocker sur le disque dur mais ce serait loin d'être optimal. En effet, les temps d'accès varient fortement avec différents types de mémoire (plus c'est rapide et plus c'est petit, et cher). On a donc une organisation de la mémoire en différents types.

4.1 Différents niveaux de mémoire

Dans un ordinateur actuel, on peut séparer la mémoire en différents niveaux :

- Tout d'abord, le processeur lui-même possède une toute petite zone mémoire, qui correspond à un ensemble de registres de 8,16 ou 32 bits (au plus quelques dizaines de registres). Ces registres sont les plus rapide, et on peut y accéder en 1 cycle d'horloge.
- Ensuite, toujours dans le processeur, on dispose de plusieurs zones mémoires (appelées caches L1, L2, L3). Pour un processeur lambda, les tailles peuvent être respectivement 128Ko, 1Mo et 6Mo, avec des vitesses qui décroissent de 700Go/s à 100Go/s. Les premiers niveaux de cache stockent généralement le data, puis un mélange de data et d'instructions, puis dès le cache L3 sont des caches généraux, partagés par plusieurs coeurs (et on a parfois des caches L4 ou L5 dans des architectures plus spéciales, où les caches sont des plus en plus partagés et de plus en plus gros mais lents). Les temps d'accès grimpent aussi : 4 cycles pour le L1, une dizaine pour le L2, et entre 40 et 100 pour le L3 (selon que la donnée soit partagée ou non).
- Après on dispose de la RAM qui est stockée en dehors du processeur, et peut prendre une centaine de cycles, voire plusieurs centaines (donc un délai de quelques dizaines de ns). Cette mémoire est cependant beaucoup plus grosses (dizaines de Go de nos jours).
- On arrive après au stockage principal, sur un SSD ou un disque dur. Les premiers sont beaucoup plus rapides, mais on a encore un bond en latence (100 microsecondes, soit plus de 1000 fois le coût d'un accès RAM). C'est cependant encore 100 fois plus rapide que la latence des disques durs.

On voit donc qu'entre une latence de 0.3ns et une latence de 10ms, on a tout intérêt à stocker les données au bon endroit.

4.2 Gestion des caches

Pour optimiser l'exécution des processus, l'OS doit donc savoir où stocker les données. Des données fréquemment utilisées doivent donc être stockées dans les registres, ou les caches de bas niveau, et les données ephémères importent peu. Mais cela pose un problème : quand on cherche une donnée, comment savoir si elle est dans le cache ou pas ? En pratique c'est assez simple, vu que les délais sont exponentiellement long, on peut se permettre de vérifier chaque cache en partant du plus bas niveau. Mais comment savoir si l'adresse est bien chargée dans le cache, à part en vérifiant tout le cache (ce qui prendrait beaucoup plus de temps). Cette question correspond à celle de l'associativité du cache. En effet, si une adresse en RAM peut aller sur n'importe quelle zone du cache, on doit tester tout le cache. Si par contre à chaque adresse de la RAM on associe une unique adresse dans chaque niveau de cache, j'ai un unique endroit à vérifier à chaque niveau. Mais comme le cache est beaucoup plus petit que la RAM, de nombreuses adresses vont correspondre à la même case du cache, et je risque de devoir faire des remplacements permanents. On fait donc correspondre à chaque case d'un niveau entre deux et huit cases d'un niveau plus bas. On a donc au plus huit cases à vérifier, et on a moins de chance d'effacer des données utiles quand on veut mettre une nouvelle donnée dans le cache. Décider ce qu'on garde dans le cache est naturellement difficile (certains algorithmes simples ne s'en sorte pas trop mal, comme garder une liste des cases du cache accédées les moins fréquemment, mais le problème est que cela engendre un nouvel overhead).

4.3 Mémoire en UNIX

Ce qui est pratique avec le cache, c'est que l'humain ne s'en préoccupe presque jamais, vu que c'est le travail du processeur, de l'OS et dans une certaine mesure des compilateurs. Avec une contrainte cependant : que se passe-t-il si on tombe à court de RAM ? Heureusement, à peu près toute installation linux a une zone dédiée du disque dur qui sert à supplanter la RAM quand elle ne suffit pas, mais les temps d'exécution chutent alors complètement. En fait, pour simplifier les choses, les programmes en UNIX ont tous accès à un adressage mémoire qui ne correspond pas à de vraies zones dans la RAM, mais est géré par le noyau. Cela a déjà un effet très important d'isolation des processus. En effet, vu qu'on accède à une zone mémoire qui ne concerne que nous et est contigüe, il n'y a aucun moyen d'accéder à la mémoire d'un autre processus. Pour gérer cela, le système implémente des **pages**, qui sont des unités de mémoire virtuelle (par exemple des pages de taille 4Ko, mais parfois beaucoup plus), contenant les variables du processus. Quand un programme est lancé (ou est à nouveau actif), le noyau charge les pages correspondantes en mémoire. Tant qu'on a peu de pages et que la RAM n'est pas pleine tout va bien, mais si la RAM se remplit on va être obligé de libérer de l'espace en mettant les pages inutilisées sur le swap. Quand le processus propriétaire en aura besoin à nouveau, cela créera des page fault, et on devra à nouveau effectuer un remplacement. Dans les cas critiques on se retrouve dans une situation de thrashing, où la majorité des ressources du système est occupée à charger des pages, qui n'ont pas le temps d'être vraiment utilisées avant d'être remplacées. Pour éviter cela, on essaye de coder en ayant une utilisation locale de la mémoire. Par exemple, si je demande d'énumérer les éléments d'un tableau par ligne puis colonne, ou par colonne puis ligne, il ne devrait pas y avoir de différence. En pratique c'est faux sauf si le tableau tient en entier dans ma page. Si le système stocke les éléments du tableau ligne par ligne, j'accèderai à toutes les données de manière séquentielles et regarderai donc les pages les unes après les autres, alors que dans l'autre cas je ferai des aller-retours constants car deux cases consécutives sur une colonne se retrouvent souvent sur des pages différentes (alors que c'est très rare sur les lignes). Il faut donc coder en faisant attention à la localité de la mémoire.

4.4 Des problèmes de prédiction

Les problèmes de prédiction de cache ont aussi lieu pour l'exécution des programmes. En effet, depuis quelques dizaines d'années, l'exécution d'un programme n'est plus complètement déterministe. En effet, à cause des différences de vitesse entre processeur et mémoire, près de deux tiers des cycles cpu peuvent être gâchés en attente de ressource. Notamment, quand on fait un branchement conditionnel (if), et que le test dépend de la mémoire, on est à priori obligé d'attendre le résultat du test. Vu que cela fait disparaître les avantages du pipelining, et que les sauts conditionnels sont souvent très courts, les processeurs modernes font de la prédiction de branchement : ils "devinent" la valeur du if, et commencent à exécuter cette branche. S'ils ont juste tout va bien, sinon il faut faire un petit retour en arrière. Cette erreur de prédiction coûte cependant de nombreux cycles, mais est difficile à prévoir. Optimiser son code sans le tester est donc généralement une mauvaise idée car cela peut se retourner contre nous[3].

[3]Voir les slides de Carine Pivoteau à ALEA 2016 http://alea2016.gforge.inria.fr/slides/Pivoteau.pdf

5 Communications inter-processus

On a vu que les processus avaient chacun accès à leur mémoire mais que celle-ci est propre à eux même et n'est pas partagée. Naturellement, on pourrait ouvrir un fichier, écrire des données dessus et laisser un autre processus le lire (en convenant à l'avance d'un nom de fichier) avant de tout supprimer mais ce n'est pas très propre. Voyons donc comment faire fonctionner des processus en tandem.

5.1 Signaux

Tout d'abord, à la création d'un processus par fork, le père reçoit le PID du fils (et le fils 0). Si le père décide ensuite de faire un wait, une communication est possible mais minimale : dans le return de sa fonction main, le fils peut ainsi indiquer un octet qui est récupéré par le wait du père. Cette méthode est assez peu utilisée, et le plus souvent pour le debugging, car elle est très limitée et ne suit pas les codes habituels sous Unix (renvoyer 0 si tout va bien).

On a aussi une autre méthode, permettant plutôt de contrôler les autres processus : les signaux. Par l'utilisation de l'appel kill, on peut envoyer un signal (représenté par une macro commençant par SIG, dont le code a 32 valeurs possibles) à un autre processus. Par défaut, ces signaux tuent généralement la cible, mais on peut créer des handler qui permettent au processus d'exécuter un code particulier à la réception du signal. Le noyau lui-même envoie aussi par moment des signaux aux processus (comme SIGSEGV qui correspond à une faute de segmentation). Quelques remarques importantes :

- Plusieurs appels ne peuvent pas être gérés : SIGKILL tue le programme immédiatement, SIGSTOP le met en pause jusqu'à l'arrivée d'un SIGCONT.
- Un seul appel de chaque type est enregistré à priori et est anonyme: si un processus reçoit plusieurs SIGSEGV de ses descendants, il ne pourra pas savoir combien lui ont envoyé ce signal (ni qui).
- Il n'y a aucune garantie sur l'ordre d'execution des différents signaux.
- Il se peut que le code de gestion d'un signal (handler en anglais) soit interrompu par l'arrivé d'un autre signal, et causer des comportements absurdes. Il faut donc faire très attention à ce qu'on y écrit (et n'y faire que des opérations atomiques, donc pas de manipulation de fichier).
- Un processus peut s'envoyer un signal à lui-même. C'est le cas de SIGALARM qui permet à un processus de se mettre un "réveil", et donc par exemple d'empêcher d'attendre indéfiniment des ressources : on créé le réveil avant de demander les ressources, et si au moment où il sonne on ne les a pas on interrompt la demande et sort de la boucle.

5.2 Tubes et partage de zone mémoire

Vu que les signaux et les valeurs de retour sont assez limités, on a besoin d'outils plus riches pour assurer la communication. Le plus simple s'appelle le tube (pipe) et permet la communication entre un père et son fils. Comme en Unix tout est fichier, les pipes ne dérogent pas à la règle. Pour en créer un on commence par créer un tableau de deux variables (int tube[2];) puis on appelle pipe(tube). On peut alors écrire et lire dans ce tube (avec les primitives write et read, comme sur un fichier normal). Plusieurs remarques sur ces tubes :

- Chaque octet écrit va être lu une seule fois : une fois lu, l'octet disparaît du tube (donc on ne peut pas utiliser un tube pour diffuser des informations à plusieurs personnes).
- La lecture peut être bloquante : si on essaye de lire trois entiers sur le tube mais que pour le moment seuls deux ont été écrits, on va être en attente que le tube se remplisse.

- L'écriture peut aussi être bloquante : comme un tube a un buffer qui n'est pas infini, si le buffer est rempli le processus voulant écrire devra attendre que quelqu'un lise le tube et libère une partie du buffer.
- Avoir plusieurs lecteurs écrivains sur un tube est dangereux vu que toutes les opérations ne sont pas atomiques dessus.

Les tubes décrits jusqu'ici sont aussi appelés tubes anonymes, et ne peuvent être utilisés que par un processus et ses descendants (qui ont accès au tube s'il a été déclaré avant le fork). Pour communiquer entre processus quelconques on utilise les tubes nommés, via la primitive mkfifo qui va créer un descripteur de fichier pouvant être accéder par n'importe quel processus (s'il a les permissions nécessaires). Pour des considérations réseau, les sockets ont aussi un fonctionnement similaire (voir le man).

Le problème avec ces méthodes est que cela ne permet pas à plusieurs processus de communiquer de manière efficace (par exemple, si on veut avoir une variable partagée, il faut que chacun tienne tous les autres au courant, et des problèmes de priorité arrivent. Pour faire cela on peut décider d'avoir un vrai fichier où on écrit nos variables et où chacun peut les lire, mais c'est à priori très coûteux si le fichier est stocké sur le disque car on doit attendre que le noyau valide chaque lecture et écriture. On peut donc faire appel à la primitive mmap qui conserve un fichier intégralement en mémoire et où les modifications ne sont répercutées sur le fichier disque que lorsqu'on le demande explicitement.

5.3 Mutex et Sémaphores

Les processus ne font pas que communiquer, et ont parfois besoin d'accéder aux mêmes ressources. Si j'essaie d'écrire sur un fichier pendant que quelqu'un d'autre essaye de lire, je risque d'avoir des problèmes. Le domaine s'occupant de ces problèmes s'appelle la théorie de la concurrence. Un problème classique est le dîner des philosophes, où 5 personnes sont à une table ronde et ont toutes besoin de deux baguettes pour manger, mais il n'y a que 5 baguettes, une entre chaque paire de voisins. Le problème dans une telle configuration est que si tout le monde décide de prendre la baguette de gauche puis la droite, et agit en même temps, ils se retrouvent tous bloqués (on appelle ça un deadlock). On doit donc trouver une manière qui permette d'éviter cette situation. Une première méthode est d'utiliser une mutex (pour **ex**clusion **mut**uelle). Ces variables partagées permettent de rendre "atomiques" ces opérations de demande de ressources. Atomique veut ici dire que l'opération marche en entier ou échoue intégralement mais n'est pas partiellement complétée. Ainsi, si un philosophe prenait deux baguettes à la fois il n'y aurait pas de deadlock. Pour obtenir cela on peut avoir un jeton au milieu de la table, et avant de prendre des baguettes chaque philosophe doit prendre le jeton, prendre ses deux baguettes s'il peut, sinon remettre les baguettes puis le jeton et attendre un peu. Ce jeton ne pouvant pas être dans les mains de deux philosophes en même temps, il garantit qu'un unique philosophe est en train de réquisitionner des ressources à chaque instant, et rend donc atomique la prise de baguette. On peut aussi utiliser des sémaphores, technique un peu différentes qui consiste à avoir non pas une variable booléenne partagée (mutex) mais un entier (le sémaphore). Quand quelqu'un veut prendre des ressources, il demande à diminuer cette variable d'une certaine quantité, et quand il y arrive récupère les ressources, les utilise, les libère, puis augmente le sémaphore. Ainsi on a un entier qui désigne le nombre de ressources libres et plusieurs personnes peuvent se servir en même temps tant que le nombre total de ressources est suffisant. Cela pourrait correspondre à mettre toutes les baguettes au milieu de la table (mais pas nécessairement, on peut aussi l'adapter au cas où chacun ne peut utiliser que les baguettes à sa droite et sa gauche). Un sémaphore est donc une généralisation du mutex qui est souvent plus efficace quand de nombreux processus demandent des ressources qui sont en assez grande quantité.

6 Transfert d'information et codes

On a vu comment les programmes d'une même machine se transmettaient des informations, mais comment faire entre plusieurs machines ? Pour faire cela, il faut distinguer plusieurs étapes : représenter le signal de manière physique, interpréter cela comme une suite de '0' et '1', et corriger les erreurs potentielles qui apparaissent, le tout le plus vite possible.

6.1 Transmettre un signal

Que l'on soit en train de communiquer par ondes radio ou à travers une interface ethernet, on transmet généralement une onde sinusoïdale que l'on module (par exemple on peut augmenter légèrement la fréquence ou la diminuer (correspondant à un '0' et un '1' respectivement), pour faire de la modulation en fréquence. On peut aussi faire de la modulation en amplitude. Dans un circuit électronique usuel, et dans les bus, un groupe de bits est envoyé en parallèle (par plusieurs connecteurs). Quand on a un unique moyen de communication on peut toujours envoyer une variété de signaux, en utilisant le multiplexage[4]. Cela correspond à faire passer plusieurs signaux sur un même canal, et plusieurs techniques sont possibles. La plus simple est la division en temps, où chaque signal à tour de rôle a le contrôle du canal pendant une courte durée (ce qui peut être difficile à synchroniser toutefois). On peut aussi avoir une division en fréquence, où les fréquences de base des différents signaux sont suffisament éloignées pour ne pas causer d'interférences destructrices. Enfin, on peut utiliser des techniques de division par polarisation, où la phase des différents signaux varie, mais généralement un ensemble de techniques de ce type est utilisé conjointement. Tout ce domaine s'appelle la modulation du signal.

6.2 Erreurs

Qui dit signal dit bruit, et perturbations. Si on module un bit et qu'une interférence arrive au même moment, on peut avoir une erreur sur le bit. Malgré ces erreurs on peut encoder le message à envoyer, c'est à dire rajouter de la redondance, pour faire baisser arbitrairement la probabilité d'erreur. Si la probabilité d'erreur sur le bit est symétrique (un '0' a autant de chance de devenir un '1' que réciproquement), on a même un théorème permettant de caractériser les débits de communication atteignables. Tout d'abord, supposons que la probabilité d'erreur est $\frac{1}{2}$. Dans ce cas si on reçoit un '1', on ne peut pas savoir si le bit de départ était un '0' ou un '1' et on ne peut rien dire. Si la probabilité d'erreur est égale à $\frac{1}{2} + a$, alors la probabilité d'avoir le bon bit en inversant le résultat est égale à $\frac{1}{2} - a$, donc on peut supposer désormais que la probabilité d'erreur est $< \frac{1}{2}$. Si la probabilité d'erreur est $\frac{1}{2} - a$, en répétant le bit n fois d'affilée et en prenant celui qui apparaît le plus on a une probabilité d'erreur qui décroit exponentiellement en n, mais cette méthode n'est pas optimale. On a en fait deux théorèmes permettant de caractériser la capacité. Si la probabilité d'erreur est 0 et la capacité du canal est C, on peut naturellement transférer à un débit C. En fait, si on définit l'entropie (binaire) $H_2(p)$ d'une probabilité p comme $-p\log_2 p - (1-p)\log_2(1-p)$, la capacité maximale d'un canal devient égale à $\frac{C}{1-H_2(p)}$, où p est la probabilité d'erreur[5]. Comme les probabilités sont rarement symétriques ou aussi simples, on a aussi une autre version[6] pour le cas d'un bruit blanc additif (d'autres versions plus spécialisées existent) :

$$C = B\log_2\left(1 + \frac{S}{N}\right)$$

Où C est la capacité, B le débit à priori, et $\frac{S}{N}$ le rapport de signal sur bruits (où les deux sont exprimés en puissance moyenne). Ces deux théorèmes affirment que l'on peut atteindre des capacités arbitrairement proches de la capacité maximale en ayant des probabilités d'erreurs arbitrairement basses. Cela permet de montrer que les méthodes de répétition de bits sont très loin d'être optimales,

[4] Le multiplexage est incontournable en communications radio, quand un émetteur veut envoyer différents messages à plusieurs récépteurs via une unique antenne.

[5] Ce résultat s'appelle le deuxième théorème de Shannon ou le noisy-channel coding theorem.

[6] Cette version, antérieure, s'appelle le théorème de Shannon-Hartley

surtout quand on a des probabilités d'erreur faibles, mais ne donne pas pour autant de méthodes permettant d'atteindre ces capacités.

6.3 Codes correcteurs d'erreurs

Vu que les méthodes naïves utilisant les codes de répétition sont loin d'être optimales, d'autres outils ont été développés qui permettent de se rapprocher de ces bornes. Ces techniques, appelées codes correcteurs d'erreurs, permettent de transformer un message de n bits en message de $n+k$ bits, avec une garantie que l'on peut corriger un certain nombre d'erreurs. Pour faire cela on associe à chaque message initial un nouveau message encodé (qui peut être totalement différent), de manière à ce que n'importe quels deux messages encodés aient au moins d bits de différence (où d dépend du type de code). À partir d'un message reçu on trouve le message encodé le plus proche (qui a le moins de bits de différence). Si on a au plus $\left\lfloor \frac{d-1}{2} \right\rfloor$ bits corrompus on pourra donc retrouver le bon message de départ. On peut aussi décider de ne pas corriger les erreurs mais simplement de les détecter (et de demander le renvoi du message de départ). Dans ce cas on peut détecter jusqu'à $d-1$ erreurs. Par exemple, le code le plus simple permet la détection d'une erreur (mais aucune correction, car $d=1$), et consiste à rajouter un bit en fin de message avec le xor des bits. Changer un bit du message fait changer le xor, donc on a bien $d=2$. Des généralisations permettant de gérer plus d'erreurs (parfois de manière probabiliste) et sont connues sous le nom de checksum, présentes à la fin des cartes bancaires ou IBAN, ou pour vérifier qu'un fichier compressé est bien correct.

Le code le plus célebre est le code (7,4) de Hamming, permettant de corriger n'importe quel erreur dans un groupe de 7 bits, en utilisant 3 bits de redondance et 4 bits de données. Pour faire cela chaque bit de redondance correspond au xor de tous les bits de données sauf 1. S'il y a une seule erreur on peut donc la retrouver. On calcule les bits de redondance des 4 bits de données reçus, et on compare aux bits de redondance reçus. une différence unique veut dire qu'un bit de redondance a été corrompu, et trouver deux ou trois différences permet de retrouver le bit de donnée corrompu. Tout cela n'est naturellement valable que si un unique bit du message a été corrompu (ou si le message n'a pas été modifié). Ce code (7,4) est très fréquemment appliqué, mais d'autres codes de Hamming existent, pouvant toujours corriger une erreur en ayant un nombre de bits de redondance logarithmique. Ainsi le code de Hamming sur 31 bits n'a besoin que de 5 bits de redondance.

D'autres codes permettent de corriger plus d'erreurs, comme le code de Golay qui permet d'en corriger 3, ou des classes de code qui permettent de créer des codes corrigeant un nombre quelconque d'erreurs. Ces dernières années, deux types sont très utilisés, les turbocodes (d'origine française) et les codes LDPC (low density parity check). Ces codes permettent d'atteindre des débits proches de la borne de Shannon, mais sont souvent beaucoup plus difficiles algorithmiquement, et prennent beaucoup plus de temps à décoder, mais offrent d'autres possibilités. Par exemple, les codes usuels sont généralement par blocs, où on découpe le message de départ en blocs de même taille et encode chacun indépendamment. Ces deux codes avancés permettent au contraire d'encoder un flux continu de données.

Ces techniques sont aussi appelés Forward Error Correcting car elles permettent de se débarasser des contraintes de renvoi des données erronées. En effet, si on a un flux de donné où l'ordre est important et dont le débit est grand devant la latence, attendre un paquet renvoyé est très coûteux et nécessite un grand espace de stockage. Par exemple, dans les communications spatiales, les débits peuvent être faibles (de quelques Mo/s à quelques bits/s), mais les latences peuvent se compter en heures (plus de 37 heures pour Voyager 1).

6.4 Applications et erreurs complexes

Si ces codes sont utilisés dans les communications, ils ont aussi de grandes applications pour le stockage de données. Par exemple, les disques durs et CDs utilisent du code de Hamming (7,4). Sur ces supports des petites erreurs sont relativement fréquentes, et augmentent avec la densité d'informations. De même, lorsqu'on veut obtenir de la redondance dans un système de stockage comme les RAID 5 ou

6 pour des disques durs, on utilise un bit de parité, qui permet de retrouver n'importe quel disque manquant à partir des autres. Le RAID 6 est basé sur des techniques similaires (avec 2 copies de redondances). Dans ce cadre on a deux types d'erreurs : suppression de bit et modification de bit, ce qui est plus complexe que le modèle précédent, mais est toujours géré par les codes turbo et LDPC (avec quelques ajustements). De même, dans de nombreux supports de communications on n'a pas des probabilités symétriques ni indépendantes, et les erreurs ont tendance à arriver en groupe (burst). Dans ce cas, une technique simple marche très bien : on envoie k paquets en même temps, en faisant de l'entrelacement (interleaving), ce qui correspond à envoyer tous les bits n°1 de chaque paquet à la suite, puis les bits n°2, et ainsi de suite jusqu'à arriver à la fin des paquets. Si une erreur en burst corrompt moins de k bits, chaque paquet n'aura qu'un bit à corriger. On peut donc "distribuer" l'erreur sur plusieurs paquets.

7 Réseau et bases internet

Avec la section précédente on a les outils pour envoyer un signal à travers n'importe canal de communication. Si on avait un cable entre chaque paire de machines sur Terre ce serait suffisant, mais le coût de cette solution (appelée clique) est quelque peu prohibitif. Il faut donc envoyer des informations à ses voisins et que ceux-ci redirigent les informations vers le destinataire. Si un groupe de k personnes sont connectées directement les unes aux autres (par un switch ethernet par exemple), cela se gère simplement. Si toutefois on a plusieurs réseaux de ce type on a besoin de protocoles plus complexes de redirection. Ainsi, le terme Internet vient d'inter-network, car cela correspond à connecter plusieurs réseaux et à rediriger les paquets entre eux. Ces paquets contiennent les donnéers à envoyer, mais aussi des données supplémentaires nécessaire pour cette redirection, comme l'adresse de leur destinataire. Une autre information est le numéro de port. Cette information permet à la machine de savoir à quel processus le paquet est destiné. Par exemple une requête HTTP arrive sur le port 80 et le navigateur sait qu'il faut demander que les paquets arrivant sur ce port arrivent vers lui (cela étant géré par le kernel). Naturellement, de nombreuses architectures sont possibles, ainsi qu'une panoplie de protocoles.

7.1 Topologie du réseau

Si on considère un réseau de n personnes – qu'on peut modéliser comme un graphe – on a besoin d'avoir au moins $n-1$ liens pour connecter tout le monde. Un client qui redirige les paquets est souvent appelé un routeur. Dans ce cas on peut avoir une structure très simple, l'étoile (où un routeur est connecté à tous les clients), mais on peut aussi avoir des arbres où les routeurs correspondent à une hiérarchie, avec une structure arborescente. Dans les deux cas supprimer un routeur déconnecte le réseau, mais dans le cas d'un arbre seul une partie des communications est coupée par rapport à une étoile. À chaque fois il y a plusieurs métriques que l'on peut utiliser pour étudier les capacités d'un réseau :

- Le nombre de liens, qu'on cherche généralement à minimiser, et qui grandit souvent de manière linéaire avec le nombre d'individus.
- La distance maximale entre deux personnes : minimisée avec une clique ou une étoile (égale à 1 et 2), maximisée avec une ligne (linéaire), et logarithmique avec un arbre. On peut aussi s'intéresser à la distance moyenne mais c'est plus complexe.
- Le débit maximal, si tout le monde envoie des message à tout le monde. Les arbres sont généralement assez bien adaptés si on fixe une limite non pas sur la capacité des canaux mais la capacité des routeurs (qui ne peuvent pas traiter les demandes de connection de tout le réseau à la fois). Dans le cas de capacités limitées sur les canaux on peut aussi avoir plus de liens (ou des liens à plus haut débit) entre les routeurs en haut de la hiérarchie (c'est le cas aujourd'hui pour internet).
- La tolérance à la panne. Dans le cas d'une étoile, si le routeur central tombe en panne le réseau entier est déconnecté, contrairement à un arbre où seule la moitié des communications s'arrête. Des structures plus redondantes existent. La plus simple est le cycle, où chaque paquet a deux chemins potentiels pour aller vers sa destination et où la panne de n'importe quelle machine n'affecte pas trop les performances (hélas déjà mauvaises). Ces pannes peuvent être ciblées ou non : une panne de 10% des machines sur internet n'affecterait pas le résau, mais une attaque ciblée sur 0.1% des routeurs pourrait faire chuter presque toute l'infrastructure d'internet.

Jongler avec ces différents objectifs est difficiles, particulièrement quand les réseaux ne sont pas complètement contrôlés et évoluent avec le temps. Des techniques comme le fait de superposer différents arbres avec des liens tous différents permet de donner de la redondance en limitant l'augmentation du coût et la distance moyenne. De manière plus poussée, on peut même utiliser des graphes expandeurs dans des situations critiques.

7.2 Routage

On a donc un ensemble de machines interconnectées qui veulent s'envoyer des paquets. Si le réseau restait fixe et ne changeait jamais, on pourrait aisément calculer les chemins optimaux que chaque paquet devrait suivre. Chaque machine pourrait donc savoir à qui envoyer chaque paquet et par qui passer pour les redirections. Une manière de calculer cela est l'algorithme de Bellman-Ford distribué. Chaque machine maintient un tableau de tous les autres membres du réseau avec la distance qui les sépare et le prochain noeud sur le chemin vers ce destinataire. Ces distances valent au départ l'infini, sauf pour les voisins du noeud. Toutes les minutes, un noeud envoie sa version cette table de routage partielle à ces voisins. Quand un noeud reçoit une table de routage, il compare pour chaque destinataire la distance reçu et la distance qu'il connaissait déjà. Si la différence est inférieure à la distance entre lui et son voisin, cela vaut le coup de passer par le voisin et il met à jour la nouvelle distance au destinataire et le fait qu'il faille passer par le nouveau voisin. Si le nombre maximum de noeuds séparant deux individus est n, il faut au plus n minutes pour que chacun sache vers qui rediriger les paquets de manière optimale. Naturellement, faire cela naïvement créé un traffic immense, mais il y a des méthodes pour améliorer cette situation (juste envoyer des tables quand elles sont mises à jour, et uniquement la partie mise à jour) et en pratique tout se passe bien.

Il existe toutefois un vrai problème avec cet algorithme : nos réseaux ne sont pas statiques mais dynamiques, et de nouveaux liens se créent et disparaissent à chaque instant. Si on crée un nouveau lien plus court entre deux machines, l'algorithme met correctement les tables à jour, mais ce n'est pas le cas si au contraire un lien disparaît ou une distance augmente. En effet, si B a le plus court chemin vers C, et le dit à A, A va savoir qu'il faut rediriger les paquets pour C vers B. Si le lien entre B et C empire soudainement, B peut avoir intérêt à passer par A, qui renverra le paquet à B et ainsi de suite. Pour éviter cela plusieurs solutions existent, comme la méthode de l'horizon éclaté, qui empêche à A de dire à B que le chemin le plus court passe par B.

7.3 Modèles TCP/IP et OSI

Créé à la fin des années 1960, ce modèle permet de gérer le traffic sur le réseau internet. Pour faire cela, chaque machine connectée au réseau se voit attribuer une adresse IP unique[7]. Cela permet à tout le monde de savoir à qui envoyer ses messages. Le modèle TCP/IP propose des protocoles et une modélisation du traffic en quatre couches, avec une encapsulation à chaque niveau permettant aux développeurs de se concentrer sur une couche uniquement.

- La couche la plus haute, couche d'application, échange directement des données avec une autre machine sans aucune préoccupation technique sur le chemin pris par les données. Les protocoles utilisés sont par exemple HTTP, FTP ou SMTP.

- La deuxième couche, couche de transport, prend les données fournies par la couche d'application et s'occupe d'assurer qu'elles arrivent à bon port. Pour faire cela elles découpent ces données en paquets, rajoutent de la redondance et des accusés de réception pour assurer le fonctionnement de la couche supérieure. Deux protocoles sont utilisés : TCP, qui garantit que les données seront bien réceptionnées (avec du renvoi de paquet si nécessaire) et qu'elles arrivent dans le bon ordre. Un deuxième protocole, UDP, coûte moins en traffic (car il n'y a pas d'accusé de réception et d'overhead), et permet l'envoi de paquets sans réelle garantie. C'est toutefois désirable quand une faible perte n'est pas grave (streaming vidéo par exemple), ou quand on désire faire du multicast (envoyer les mêmes données à de nombreuses personnes).

- La troisième couche, couche réseau, s'occupe du routage et de l'organisation du réseau. Le protocole le plus utilisé est le protocole IP. Les paquets de la couche transport sont eux-même encapsulés, et on y rajoute l'adresse IP du destinataire, d'autres informations (comme le numéro

[7] Le changement d'IPV4, qui permet 4 milliards d'adresse, à IPV6 qui en permet plus de 10^{28} n'est pas encore terminé (alors que IPV6 est bien défini depuis 1998, avec des grands efforts de transition depuis 2008).

de port) et on les envoie. Cette couche n'assure aucune garantie sur la réception des paquets (c'est le rôle de la couche précédente). Cette couche gère aussi les requêtes de type ping, et le tri des paquets (par exemple un pare-feu peut empêcher la réception de paquets provenant d'une catégorie d'adresse IP, ou bien utilisant un port donné).

- La couche la plus basse, appelée couche data link, s'occupe directement des échanges physiques entre deux machines à travers un cable. Les protocoles utilisés sur cette couche permettent par exemple de savoir quelle est l'adresse des autres machines où nous sommes connectés.

Grâce à cette séparation en plusieurs couches, les développeurs de la couche d'application peuvent se concentrer sur leur contenu et savent que leurs paquets vont arriver à bon port. Un autre modèle existe aussi, postérieur et moins fréquemment utilisé, sépare encore plus les rôles des couches, en scindant la couche data link en deux (une partie dédiée à la transmission de bits bruts, et l'autre à la transmission de paquets de manière correcte entre deux extrémités d'une connection). Elle divise aussi la couche data en application en 3, avec une couche de gestion des sessions, une couche gérant la compression et le chiffrement, et enfin une couche d'application pure.

Ces deux modèles sont naturellement une abstraction, mais ils permettent de se représenter assez bien le rôle des différents protocoles.

7.4 Structure d'internet

Pour assurer le fonctionnement du réseau global, il est important de connaître un certain nombre de services. Tout d'abord, vu que les adresses IP ne sont pas très facile à gérer, pour le world wide web on préfère utiliser des adresses en textes, appelées noms de domaine. Ainsi, je peux acheter (auprès d'un revendeur accrédité, dépendant de l'ICANN) le nom koliaza.com. Une fois cela fait, je demande[8] que tous les accès à cette adresse soient redirigés vers une adresse IP de mon choix. Cette redirection est faite grâce à des serveurs DNS (pour domain name system). Ainsi, une requête http va d'abord faire une demande au serveur DNS pour avoir l'IP de la destination, et quand cette requête est satisfaite, envoyer la demande originale à la bonne adresse IP. Le traffic internet est complexe à analyser, et est constitué d'un mélange de fournisseurs qui possèdent des cables et les louent, de zones d'échanges entre différents réseaux (correspondant à différents fournisseurs) et est paradoxalement parfois assez fragile (un cable de fibre optique coupé en Géorgie a déconnecté internet en Arménie pour 5 heures en 2011).

[8] C'est fait automatiquement par le vendeur de noms de domaine comme godaddy ou ovh.