

Unix au temps du couvre-feu (3)
Compléments et exercices

Thierry Dumont *pour* l'ALDIL

Table des matières

1. Chemins dans l'arbre des fichiers.	4
2. La commande <code>ls</code>	5
3. Quelques autres commandes	6
4. Plomberie	7
5. Quelques petits trucs à savoir à propos du shell	8
6. Utilisateurs et groupes	8
7. Droits	10
8. Processus	12
9. Forme (presque générale) des commandes Unix	15
10. Noms de fichiers, méta-caractères, et expressions régulières	15
11. Deux manières d'acquérir des pouvoirs exceptionnels	17
12. Le shell : configuration, environnement	19
13. Quelques fichiers et répertoires cachés importants	20
14. Une liste de commandes Unix	21
15. Bibliothèques (libraries)	21
16. Interface graphique, bureau etc.	23
17. Les liens	24
18. Systèmes de fichiers	25
19. Une application complète : sauvegarde d'un répertoire avec <code>rsync</code>	29
20. Réseau : le modèle en couches de l'Open Systems Interconnection (OSI)	32
21. Réseau, configuration classique : machines derrière une « box »	33
22. Comment explorer tout ça sous Linux ?	39
23. Installation de « packages », mises à jour.	42
24. Installation de packages fournis par la distribution	42
25. Installer depuis la « source »	43
26. Problématique	47
27. Un exemple : <code>ssh</code> (Secure Shell)	49
28. Cryptographie : méthodes	50
29. Est-ce que je communique avec le bon interlocuteur ?	53
1. Bibliothèques de calcul	59
2. Le COW (Copy On Write)	61
3. Arbres binaires équilibrés (Balanced Trees, B-Trees)	62
4. Une autre « source » de méthodes pour la cryptographie : le logarithme discret	65
Annexe. Bibliographie	69

Dans la suite la syntaxe :

`cmd>commande arguments`

désignera une commande (à taper dans le shell) ; le « prompt », c'est à dire le début de la ligne de commande peut varier d'une configuration à l'autre ; `cmd>` représente ce prompt et n'est pas à taper.

1. Chemins dans l'arbre des fichiers.

Dans la suite j'emploie le mot *directory* ou le mot *répertoire* pour désigner la même chose.

Il y a plusieurs moyens de définir des chemins. On rappelle que la racine de l'arbre est désignée par `cmd>./`.

- (1) Chemin absolu en partant de la racine ; exemple : `cmd>/etc/firefox/pref`

Ce qui peut se lire de droite à gauche : `cmd>pref` est dans le répertoire `cmd>firefox` et `cmd>firefox` est dans le répertoire `cmd>etc`, qui est à la racine¹, ou bien de gauche à droite : `cmd>etc` est à la racine, `cmd>firefox` est dans `cmd>etc` et `cmd>pref` est dans le répertoire `cmd>firefox`.

- (2) Chemins relatifs : le point `cmd>.` désignant le répertoire courant, les deux points `cmd>..` désignant le répertoire père et `cmd>~` (tilde) désignant le *home directory*, on peut définir des chemins *relatifs* (au-dessus, à coté, etc.), comme par exemple `cmd>../truc`.

On rappelle les deux commandes `cmd>pwd` et `cmd>cd` :

— `cmd>pwd` (print working directory) : vous dit où vous êtes.

— `cmd>cd` pour changer de répertoire (change directory) :

`cmd>cd` sans argument vous ramène à votre home-directory. Sinon, la commande doit être

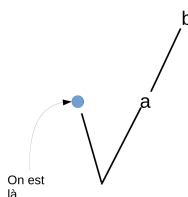
`>cd chemin`

où `> chemin` est un chemin, relatif ou absolu ; exemple :

`cmd>cd /usr/local/bin` (chemin absolu : on décrit le chemin depuis la racine), ou bien

`cmd>cd ../truc/toto` (c'est à dire aller dans le répertoire père puis là, aller dans `truc` puis dans `toto` –qui doivent exister pour que ça fonctionne–) : c'est un chemin relatif (on décrit le chemin depuis là où on est).

Illustration (déplacement avec un chemin relatif) :



Après `cmd>cd ../a/b`, on est en b.

1. Noter que dans `cmd>/etc/firefox/pref`, `cmd>pref` peut être un répertoire ou un fichier, mais si je tape `cmd>/etc/firefox/pref/`, alors `cmd>pref` est obligatoirement un répertoire.

Exercices : à faire depuis un terminal².

(1) Déplacements absolus et relatifs :

- aller directement dans `cmd>/etc/network`
- de là, aller dans `cmd>/etc`
- revenir dans le home directory.

(Note : vous avez certainement du faire un déplacement relatif).

(2) À quoi correspond : `cmd>cd ../..` ?

2. La commande `ls`

Regarder et jouer avec la commande `cmd>ls` permet de se familiariser avec les autres commandes qui fonctionnent toutes sur le même principe :

(1) `cmd>ls`

sans option (**tester**).

(2) `>ls chemin`

par exemple :

`cmd>ls /usr/bin`

Exercice : jouer avec différents types de chemins utilisés ci-dessus.

(3) `cmd>ls` avec options. Les options commencent par `cmd>-` (c'est le tiret, c'est à dire le signe "moins" du 6).

- Placez vous d'abord dans le home directory, puis tapez :

`cmd>ls -a`

Avec l'option `cmd>-a ls` montre les fichiers et répertoires cachés (installés par vos applications, leur nom commence par un point) (**tester**).

- Avec l'option `cmd>-l` :

`cmd>ls -l`

on obtient plein de renseignements (on verra ça plus tard) (**tester**).

- Combiner plusieurs options

`cmd>ls -al`

et puis aussi `cmd>ls -alt`

(résultats triés par dates) et on peut aussi inverser l'ordre de tri :

`cmd>ls -altr` (**tester**).

- Appliquer cette commande avec des options à un chemin :

`cmd>ls -alt /var/log`

etc, etc. (**tester**).

Noter que la syntaxe des options est assez souple : elles peuvent être mises dans n'importe quel ordre ; de plus au lieu de `cmd>ls -alt /var/log` on aurait pu écrire

`cmd>ls -l -a -t /var/log`

mais en ne voit pas trop quel en aurait été l'intérêt.

2. on dira plutôt « shell » que terminal, c'est plus chic.

3. Quelques autres commandes

- Les systèmes Unix contiennent leur documentation (manual). La commande est `cmd>man`.

Exercice : jouer avec `cmd>man` par exemple :

```
cmd>man ls
```

ou bien : `cmd>man man`

Le résultat peut être indigeste !


- `cmd>whoami`

Vous donne votre *login*. En principe votre home-directory est `/home/` + le résultat de cette commande. **Le vérifier.**

- `cmd>less` permet de voir un fichier page par page (appuyer sur espace pour aller à la page suivante). On peut aussi sauter à la prochaine occurrence d'un motif (taper `/motif` quand on est dans less). (`q` pour quitter less). Un exemple (**tester**) :

```
cmd>less /etc/services
```

- `>rm chemin`

Efface un fichier (celui qui est au bout du chemin).  **Attention, on travaille sans filet ! Quand un fichier est effacé, on n'a plus qu'à aller chercher la sauvegarde (si on en a une).**

Exemples :

```
cmd>rm toto
```

```
cmd>rm ~/truc/machin/toto
```

- `>mkdir chemin`

Crée un répertoire (il ne doit pas déjà exister). Exemples :

```
cmd>mkdir MonDir
```

```
cmd>mkdir /truc/machin/chose
```

Ça fonctionnera seulement si `/truc/machin/` existe déjà. Sinon faire :

```
cmd>mkdir -p /truc/machin/chose
```

et on crée comme ça tous les répertoires.

- `>rmdir chemin`

Efface un répertoire **vide**. Exemple :

```
cmd>rmdir ~/truc/machin/chose
```

Et si le répertoire n'est pas vide ? On a un message d'erreur. Alors il faut utiliser la commande `cmd>rm` avec l'option `cmd>-r` (récursive) qui efface le répertoire et tout ce qu'il contient :

```
cmd>rm -r ~/truc/machin/chose
```

Si on a peur, on peut ajouter `cmd>-i`, qui vous demandera de confirmer chaque effacement :

```
cmd>rm -ri ~/truc/machin/chose
```

- Filtre : `cmd>grep`

`cmd>grep udp /etc/services` va lister toutes les lignes qui contiennent `udp` dans le fichier `/etc/services`.

`cmd>grep -v udp /etc/services` va lister toutes les lignes qui ne contiennent **pas** `udp`.

Si on rajoute `cmd>-i` (par exemple `cmd>grep -v nagios /etc/services`) la recherche ne tient plus compte de la casse (minuscules ou majuscules).

Exercice :

La commande :

```
cmd>touch toto
```

crée un fichier vide de nom `cmd>toto`.

Créer des répertoires (emboîtés), créer des fichiers vides dedans et tout effacer³.

4. Plomberie

Deux définitions :

- (1) **Entrée standard** : ce que les commandes lisent. Par défaut, c'est votre clavier.
- (2) **Sortie standard** : là où les commandes écrivent : la fenêtre courante par défaut.

Alors :

- On peut rediriger la **sortie standard** vers un fichier :

```
cmd>ls /var/log >toto
```

Les résultats vont dans `cmd>toto` plutôt que sur l'écran. Regarder ensuite le fichier `cmd>toto` avec :

```
cmd>less toto
```

- Avec `cmd><` c'est l'entrée standard qu'on redirige depuis un fichier (on verra ça plus tard).
- Le « pipe » :

Le principe est le suivant : étant donné des commandes `cmd>A` et `cmd>B`, on fait en sorte que la sortie standard de `cmd>A` soit l'entrée standard de `cmd>B` :

```
cmd>A|B
```

(le caractère `cmd>|` est obtenu par `Alt Gr` et la touche "6").

Exemple (à tester) :

- On commence par `cmd>grep tcp /etc/services` qui liste les lignes qui contiennent `tcp` dans `/etc/services`. Évidemment, le résultat sort sur l'écran (la sortie standard).
- La commande `cmd>grep`, s'il n'y a pas de chemin défini, lit sur l'entrée standard. Donc

```
cmd>grep tcp /etc/services | grep Protocol
```

va :

- (1) Filtrer les lignes de `cmd>/etc/services` qui contiennent `tcp`.
- (2) Ensuite, `cmd>grep Protocol` récupère ce qui normalement va sur l'écran et filtre les lignes qui contiennent `Protocol`.

et voilà. On peut chaîner autant de commandes qu'on veut. Par exemple, la commande :

`>n1 chemin vers un fichier`

numérote les lignes d'un fichier (regarder par exemple ce que donne `cmd>n1 /etc/services`) ; en l'absence d'argument, la commande `cmd>n1` lit l'entrée standard.

On peut donc chaîner 3 commandes (`cmd>grep`, `cmd>grep` et `cmd>n1`) :

```
cmd>grep tcp /etc/services | grep Protocol|n1
```

et **compter** ainsi le nombre de lignes du fichier `cmd>/etc/services` qui contiennent `tcp` et `Protocol`.

Et puis on peut finir par la commande `cmd>tail` ; ainsi

3. Mettez donc l'option `cmd>-i` quand vous utilisez `cmd>rm`.

```
cmd>tail /etc/services
```

lit le fichier `/etc/services` et en montre la fin, mais `cmd>tail` sans chemin vers un fichier lit sur l'entrée standard. Donc, pour savoir combien de lignes dans `/etc/services` contiennent `tcp` et `Protocol`, et pour limiter le nombre de lignes qui, à la fin, sortent sur l'écran, on peut taper (4 commandes chaînées) :

```
cmd>grep tcp /etc/services|grep Protocol|nl|tail 4
```

5. Quelques petits trucs à savoir à propos du shell

Le shell, c'est le programme qui interprète vos commandes. En fait, il existe plusieurs shells disponibles, mais le plus populaire est celui de **Steve Bourne**, qui a connu plusieurs évolutions, et qui s'appelle maintenant « **Bourne Again Shell** ⁵ », soit `cmd>bash`. Si vous tapez :

```
cmd>which bash
```

vous verrez où il est installé (`cmd>which` vous donne le chemin vers une commande).

Il existe d'autres shells, comme `cmd>zsh`, mais bon...

Des trucs bien pratiques à savoir :

- La complétion automatique :

C'est plutôt puissant. On utilise le caractère **Tab**.

Essayez :

aTab (deux caractères : le **a** et le **Tab**),

puis :

apTab (note : il faut parfois taper 2 fois **Tab**).

On voit toutes les commandes accessibles qui commencent par **ap**.

De plus, certains logiciels introduisent des règles de complétion. Par exemple :

```
cmd>evince toto.Tab
```

proposera de visualiser le fichier `toto.pdf` s'il existe.

- L'historique :

- à titre d'exemple, `cmd>!evin` relance la dernière commande qui commençait par `cmd>evin` (attention, tout de même : pensez aux dégâts possible de `cmd>!rm`).

- `cmd>history` donne l'historique des commandes, numéroté. On peut par exemple relancer la 135^e commande en tapant :

```
cmd>!135
```

cet historique persiste, même si vous redémarrez la machine.

On peut aussi se servir des flèches pour naviguer dans l'historique.

Un bon conseil : **testez, retestez !**

6. Utilisateurs et groupes

6.1. Les utilisateurs. Les utilisateurs sont définis dans le fichier `/etc/passwd`

Regardons ⁶ une ligne du fichier, correspondant à la description de l'utilisateur de nom `moi` :

```
moi:x:1000:1000:moi,,:/home/moi:/bin/bash
```

On y voit des champs séparés par « : » qui sont :

- (1) Le nom de connexion de l'utilisateur (« login ») (`moi`, ici).
- (2) `x` : qui contenait jadis le mot de passe.

4. ou `cmd>grep tcp /etc/services|grep Protocol|nl|tail -1` cf. `cmd>man tail`

5. ah, ah, l'astuce !

6. `cmd>less /etc/passwd` (par exemple).

- (3) 1000 : en pratique, c'est un nombre qui identifie l'utilisateur, et pas son nom (plus rapide, plus simple). Ici, 1000 et `moi` sont associés et identifient le même utilisateur. 1000 est l'uid de `moi`. Et l'uid est évidemment unique, des utilisateurs différents ont des uid différents.
- (4) 1000 : identifiant du groupe de l'utilisateur.
- (5) `moi,,` : le « vrai » nom (j'aurais pu mettre `Thierry Dumont`), suivi de commentaires optionnels.
- (6) Le home directory (`/home/moi`).
- (7) Le chemin vers le shell utilisé, ici c'est `bash`.

Les utilisateurs appartiennent à des groupes (un ou plusieurs). Dans le fichier `/etc/passwd` on décrit le groupe *principal* auquel l'utilisateur appartient. *Dans beaucoup de distributions Linux, le groupe principal a le même nom que l'utilisateur (moi, ici) et ne contient qu'un seul utilisateur ! Mais rien n'empêche de regrouper les utilisateurs dans différents groupes, ce qui, on va le voir permettrait de définir des ensembles d'utilisateurs plus ou moins privilégiés par exemple.*

Remarques :

- En parcourant le fichier `passwd`⁶, on remarque que pour certains utilisateurs, le shell est `/usr/sbin/nologin` ou `false` : on ne peut pas se connecter au système sous ses noms d'utilisateur, mais on en verra l'intérêt quand on regardera les *processus*.
- Commandes :

`cmd>adduser` permet d'ajouter un utilisateur,
`cmd>deluser` d'en supprimer un.

Vous ne pourrez pas utiliser ces commandes sous votre login habituel : l'explication viendra par la suite.

Un utilisateur pas comme les autres : `root`.

`root:x:0:0:root:/root:/bin/bash`

`root` a tous les droits ! même de faire

`cmd>rm -f /`

Il convient donc *d'être très prudent* quand on prend cette identité, ce qui est absolument nécessaire pour accéder à certaines parties du système, configurer, installer des logiciels etc. On regardera ça par la suite.

6.2. Les groupes (d'utilisateurs). Ils sont décrits dans le fichier `/etc/group` qui pour chaque groupe donne :

- Le nom du groupe.
- `x` : (champ désactivé).
- Le group-id (gid), un nombre unique.
- La liste des utilisateurs qui appartiennent à ce groupe.

Par exemple, sur ma machine, le fichier `/etc/group` contient la ligne :

`lpadmin:x:121:moi.`

Ici, l'utilisateur "moi" appartient au groupe `lpadmin`. On peut conjecturer que "lpadmin" a à voir avec les imprimantes (*line printer administration*) et qu'être membre de ce groupe va donner des droits d'administration particuliers.

Exercice : À quels groupes est ce que j'appartiens en tant qu'utilisateur ? (utiliser `cmd>grep` et `cmd>/etc/group`⁷).

⁷. Ou plus simplement la commande `cmd>groups` (sans argument).

7. Droits

7.1. Commençons par un petit essai. Dans le fichier `/etc/passwd`, le champ du mot de passe est remplacé par un `x`, le mot de passe effectif étant dans `/etc/shadow`.

Exercice : Essayez de regarder le fichier `/etc/shadow` (`cmd>less /etc/shadow`). Que se passe-t-il ?

Il y a visiblement des problèmes de permission (droits) : c'est normal, car on a affaire à un fichier *sensible*, mais comment est-ce que ça marche ? Il faut s'intéresser aux *droits*.

7.2. Droits concernant les fichiers, les répertoires etc.

Résultat de :

```
cmd>ls -l /etc/passwd
```

```
-rw-r--r-- 1 root root 3444 déc. 6 11:42 /etc/passwd
```

Le premier tiret de

```
-rw-r--r-- 1 root root 3444 déc. 6 11:42 /etc/passwd
```

indique qu'on a affaire à un fichier ; à la place du `-` on aurait `d` pour un répertoire.

Ensuite, il y a 3 triplets de 3 caractères (ici `rw-`, `r--` et `r--`). Dans chaque triplet le premier caractère peut être `r` ou `-`, le second `w` ou `-` et le troisième `x` ou `-`.

- `r` indique le droit de lire, `-` l'impossibilité de lire.
- `w` indique le droit d'écrire, `-` l'impossibilité d'écrire. Attention, écrire doit être pris au sens le plus large : modifier, déplacer dans l'arborescence, renommer ou effacer, tout ce qui peut modifier.
- `x` indique :
 - pour un fichier, le droit d'exécuter, c'est à dire que le fichier est une commande, un programme exécutable.
 - pour un répertoire, c'est le droit de traverser le répertoire, ce qui n'implique pas de lire son contenu !

Et `-` indique l'absence de ce droit.

Que signifient chacun de ces 3 triplets ?

- (1) Le premier, les droits de l'utilisateur (`root` dans notre cas). Qui, ici, peut donc lire le fichier et écrire dessus (modifier, effacer, déplacer, etc !!!).
- (2) Le second, les droits des autres membres de son groupe (en pratique, dans ce cas précis, il n'y en a sûrement pas) : seulement lire dans le cas présent.
- (3) Le troisième : les droits du reste du monde, c'est à dire de tous les autres utilisateurs : ici, c'est seulement le droit de lire.

Peut-on changer les droits d'un fichier ou d'un répertoire ? Oui, à condition qu'il vous appartienne, dans ce cas la commande est `cmd>chmod` ; on va en voir un exemple plus loin.

7.3. Autres exemples.

- Pour `/etc/shadow`, les droits sont :

```
-rw-r----- 1 root shadow 1763 déc. 6 11:42 /etc/shadow
```

et donc le fichier appartient à `root` et au groupe `shadow`. Vu les droits, on voit qu'un utilisateur qui n'est pas "`root`" et qui n'appartient pas à "`shadow`" ne peut pas lire ce fichier. Notons que si on ajoutait un utilisateur comme "`moi`" au groupe "`shadow`", alors il pourrait lire ce fichier.

— **bash** : le shell. Pour savoir où il est, la commande est :

```
cmd>which bash
```

Le résultat est probablement `/usr/bin/bash`. Ensuite

```
cmd>ls -l /usr/bin/bash
```

donne :

```
-rwxr-xr-x 1 root root 1183448 juin 18 2020 /usr/bin/bash
```

Remarquez les “x” pour tout le monde : c’est normal, car il s’agit d’un programme utilisé par tout le monde.

Exercice : Étudiez la racine du système de fichiers en tapant : `cmd>ls -l/`

Exercice :

(1) `cmd>cd /tmp` (pour être dans un coin tranquille).

(2) Créez des répertoires emboîtés :

```
cmd>mkdir toto
```

```
cmd>mkdir toto/tutu8
```

Créez un fichier vide dans tutu de nom “file” (ou ce que vous voulez) :

```
cmd>touch toto/tutu/file
```

(3) Pour voir les droits affectés à toto : `cmd>ls -dl toto`. Cela devrait donner (en remplaçant moi par votre nom d’utilisateur) :

```
drwxrwxr-x 3 moi moi 4096 déc. 24 15:23 toto
```

(4) Vérifiez que les commandes :

```
cmd>ls toto et cmd>ls toto/tutu
```

fonctionnent.

(5) Maintenant changez les droits de toto :

```
cmd>chmod chmod ugo-rw toto
```

(ce qui veut dire “enlever les droits de lire et d’écrire à l’utilisateur (u), au groupe (g) et aux autres (o)).

```
cmd>ls -dl toto
```

vous indique les nouveaux droits :

```
d--x--x--x 3 moi moi 4096 déc. 24 15:23 toto
```

On ne peut plus lire ni écrire.

(6) Regardez maintenant ce que donnent les commandes :

— `cmd>cd toto` puis `cmd>ls toto`.

— Maintenant que vous êtes dans `toto`, tapez `cmd>cd tutu` ou `cmd>cd /tmp/toto/tutu`. Vérifiez que vous pouvez écrire dans tutu (`cmd>touch xx` par exemple).

(7) Retour dans `/tmp` : `cmd>cd /tmp`. Essayez d’effacer `toto`. Comme il n’est pas vide, la commande `cmd>rmdir` ne peut pas fonctionner. Il faut utiliser `cmd>rm -r toto`. Que se passe-t-il ?

(8) Redonnons nous le droit d’écrire sur et dans `toto` : `cmd>chmod u+w toto`. Essayons à nouveau `cmd>rm -r toto`.

Pourquoi est ce que ça ne marche pas ? Parce que `cmd>rm -r` a besoin de lire le contenu de `toto` pour l’effacer. On tape `cmd>chmod u+r toto` et là, la commande `cmd>rm -r toto` efface tout. Subtil !

8. On peut faire ça en une seule commande `cmd>mkdir -p toto/tutu`

8. Processus

Dans un guide Unix/Linux on peut lire : « *Un processus est une instance d'un programme en train de s'exécuter, une tâche. Le shell crée un nouveau processus pour exécuter chaque commande* ». Le terme *instance* faisant partie du sabir informatique (le mot n'est pas français), on va tenter d'illustrer ça :

```
cmd>which firefox
```

Réponse :

```
/usr/bin/firefox.
```

Le programme firefox est donc stocké dans le fichier `/usr/bin/firefox`.

Que se passe-t-il si je tape `cmd>firefox` en ligne de commande ? Le shell (bash) va chercher⁹ un fichier de nom `firefox`¹⁰ et, après l'avoir trouvé, et de manière un peu simplifiée, demander au noyau linux de le copier en mémoire et de lancer son exécution.

À l'exécution, ce n'est pas seulement une copie qui réside en mémoire, mais aussi des données¹¹, une description des fichiers ouverts et un « état programme », qui pointe vers l'instruction en cours dans le programme.

À partir de ce moment là, cette "copie" (= instance) devient pendant toute la durée de son exécution un **processus**.

Le processus ne peut pas accéder en dehors des zones de mémoire que le système lui a alloué.

À chaque processus sont associés :

- (1) un numéro (PID, *process id*) unique et chaque processus appartient à un utilisateur et hérite des droits (de lire, d'écrire,...) de l'utilisateur *et* des groupes auxquels il appartient. D'autre part, un utilisateur n'a aucun droit sur les processus des autres utilisateurs.
- (2) un processus *père*, repéré par son PPID (*parent process identifier*) : *tout processus* a un père, à une exception près, car il faut bien un ancêtre commun : c'est le processus `init` qui a le PID 1. Tous les autres en descendent et ont des PID > 1.

Pour pouvoir faire quelques expériences, il faut d'abord apprendre à maîtriser l'outil qui permet de voir quels sont les processus présents et leurs caractéristiques : la commande `cmd>ps` suivie d'options.

8.1. La commande `cmd>ps`. La commande `cmd>ps` est un peu bizarre.

- Si vous tapez juste `cmd>ps`, vous ne verrez que les processus qui dépendent de votre shell (terminal) soit en général juste `bash` et `ps` (puisque justement, vous êtes en train de faire tourner...`ps`) :

```

PID TTY          TIME CMD
338023 pts/5        00:00:00 bash
371762 pts/5        00:00:00 ps
```

mais vous voyez quand même les PID, le terminal TTY auquel ils sont attachés, le temps calcul dépensé TIME et le nom de la commande CMD.

- Si vous tapez `cmd>ps -e`, vous voyez tous les processus en cours. Je recommande de taper

```
cmd>ps -e | less
```

car il y a beaucoup de processus en cours ! Mais ça ne dit quand même pas grand chose sur chaque processus.

9. dans une liste de chemins bien définis – on verra ça plus tard –

10. il y a forcément plein de `x` dans les droits du fichier `/usr/bin/firefox` : voir plus haut !

11. Pensez par exemple à `libreoffice` : les données, c'est le texte que vous écrivez.

- `cmd>ps -F -U moi` (remplacez `cmd>moi` par votre nom d'utilisateur) : vous montre tous vos processus en vous disant beaucoup de choses.

```
cmd>ps -F -U moi | less
```

ou

```
cmd>ps -F -U moi | head12
```

vont vous permettre de voir le début de cette liste. La première ligne est :

```
UID    PID    PPID  C    SZ    RSS  PSR  STIME  TTY    TIME  CMD
```

Détaillons un peu :

- `UID PID PPID` : on a vu ça plus haut.
 - `C` : pourcentage d'utilisation du processeur.
 - `SZ RSS` : la taille mémoire réservée par le processus et la taille réellement utilisée.
 - `STIME` : date de démarrage du processus
 - `TTY` : le terminal depuis lequel la commande a été lancée. Mais si la commande n'a pas été lancée depuis un terminal, on a un "?".
 - `TIME` : le temps UC consommé par le processus depuis son démarrage.
 - `CMD` : la commande.
- Avec

```
cmd>ps -aef
```

on voit de manière étendue les processus de tous les utilisateurs.

Exercice : Commencez par agrandir au maximum votre fenêtre de ligne de commande (en général la touche F1 le fait (F1 aussi pour revenir à un terminal de taille normale)).

Ensuite :

```
cmd>ps -aef | less
```

Vous devriez voir quelque chose qui ressemble à ça :

UID	PID	PPID	C	STIME	TTY	TIME	CMD
root	1	0	0	déc.13	?	00:00:12	/lib/systemd/systemd --system --deserialize 16
root	2	0	0	déc.13	?	00:00:00	[kthreadd]
root	3	2	0	déc.13	?	00:00:00	[rcu_gp]
root	4	2	0	déc.13	?	00:00:00	[rcu_par_gp]
root	6	2	0	déc.13	?	00:00:00	[kworker/0:0H-kblockd]
root	9	2	0	déc.13	?	00:00:00	[mm_percpu_wq]
root	10	2	0	déc.13	?	00:00:07	[ksoftirqd/0]
root	11	2	0	déc.13	?	00:07:10	[rcu_sched]
root	12	2	0	déc.13	?	00:00:02	[migration/0]

Ce qui s'analyse ainsi :

- Le process 1 est `systemd` qui va lancer... beaucoup de choses. Sur d'autres distribution (la mienne est une Ubuntu 20/10) cela peut être `init` (c'est très « Linux canal historique »).
- Si on regarde les processus suivants, on voit qu'ils ont des PID croissants, mais qu'ils sont lancés (PPID) par le processus de PID = 2.
- Supposons que firefox est lancé (sinon, il n'y a qu'à le lancer!).

```
cmd>ps -aef | grep firefox
```

On obtient à peu près (la sortie est tronquée à droite¹³) :

12. `head` montre tes premières lignes d'un fichier. De même, il ya `tail`...

13. et chez vous, ça va forcément être un peu différent.

```

moi      373641  337755  6 16:58 ?      00:00:13 /usr/lib/firefox/firefox
moi      373755  373641  2 16:58 ?      00:00:06 /usr/lib/firefox/firefox ...
moi      373801  373641  0 16:58 ?      00:00:00 /usr/lib/firefox/firefox ..
moi      373956  373641  1 16:59 ?      00:00:02 /usr/lib/firefox/firefox ...
moi      374054  373641  3 17:00 ?      00:00:04 /usr/lib/firefox/firefox ...
moi      374100  373641  2 17:00 ?      00:00:03 /usr/lib/firefox/firefox ...

```

Ce qui s'analyse ainsi (faire la même analyse chez vous) :

- (1) Il y a 6 processus. les deuxième et les suivants ont un PPID = 373641, et donc ils ont été lancés par le processus de PID = 373641, qui est le premier de la liste¹⁴.
- (2) Mais si on regarde la première ligne, on voit aussi que le processus de PID = 373641 au PPID = 337755. Qui donc a lancé ce "premier" firefox ?

```
cmd>ps -aef |grep 337755
```

et vous aurez son "père"¹⁵.

8.2. D'autres commandes associées aux processus.

8.2.1. *Voir les processus tourner* : La commande `cmd>top` vous montre les processus actifs en tête. On peut jouer sur le taux de rafraîchissement, et tuer un processus (`k`).

8.2.2. *Arrêter un processus : la commande `cmd>kill`*. Cette commande permet d'envoyer un signal à un processus (en pratique, on tue le processus). La syntaxe est : `kill` suivi d'au moins un espace et le PID du process à tuer¹⁶.

Exercice : Chercher à nouveau les processus firefox comme ci-dessus, et tuer le premier. Chez moi cela donne :

```
cmd>kill 373641
```

En procédant ainsi tue le père... et ses descendants => plus aucun firefox ne tourne. On peut aussi tuer un des ses fils.

8.3. Autres notions sur les processus.

- (1) Espace utilisateur et espace noyau : la mémoire est coupée en deux parties, l'espace utilisateur et l'espace noyau. Tous vos processus tournent dans l'espace utilisateur, et ceux du noyau dans l'espace noyau. Vous ne pouvez pas intervenir sur les processus qui tournent dans l'espace noyau : c'est une sécurité supplémentaire. Avec les CPUs Intel, ces protections sont mêmes assurées par le processeurs (les *rings*).
- (2) Un processus peut créer des processus fils. On l'a déjà vu avec `initd`. Une notion un peu différente est le *fork* ou un processus crée une copie de lui même.
- (3) Les processus peuvent communiquer entre eux. Il existe pour cela différents mécanismes (exemples : une zone de mémoire partagée, les sockets (une sorte de réseau interne à la machine), le *pipe* qu'on a vu avec la commande `|` etc.
- (4) Un processus peut créer des *processus légers* appelés *threads*. La différence avec la création de processus classiques, c'est principalement que tous les *threads* et leur père partagent la même zone mémoire. Si la commande `cmd>top` vous montre un programme qui consomme plus de 100% d'UC, c'est que plusieurs threads tournent en parallèle. Vous pouvez installer le package `htop` qui permet de bien visualiser ça. L'intérêt des *processus légers* est la rapidité de la communication entre eux puisqu'il n'y a aucun mécanisme

14. Pourquoi firefox lance-t-il autant de processus ? probablement pour des questions de performances et/ou de sécurité.

15. chez moi, c'est un processus de `xfce4`, car j'utilise `xfce`, pas `gnome`.

16. Pour les processus récalcitrants, on peut faire `cmd>kill -9 PID`.

de partage de données à mettre en jeu¹⁷. Le défaut, c'est que leur programmation est délicate (partage de mémoire => pas de protection d'accès mémoire entre threads).

- (5) **Les démons (daemons)** : Ce sont des programmes résidents qui sont à priori chargés au démarrage. Quelques exemples : **cupsd** : gestionnaire d'impression, **rsyslogd** : gestion des "logs", **ntpd** : synchronisation de l'horloge avec une horloge distante, etc.

9. Forme (presque générale) des commandes Unix

À titre d'exemple, quelques unes des différentes formes que peut prendre la commande **ls** :

- (1) `cmd>ls`
- (2) `cmd>ls /tmp`
- (3) `cmd>ls /tmp /usr/ /bin`
(ou `cmd>ls /var/log/*.log`, on verra ça plus loin).
- (4) `cmd>ls -l /tmp /usr/ /bin`
- (5) `cmd>ls -l -t -r /tmp`
mais en pratique on tapera plutôt :
`cmd>ls -ltr /tmp`

On voit la structure d'une commande (les crochets { } indiquent quelque chose de facultatif) :

`commande {-options} {-options} {arguments}`

Une description plus exacte de la syntaxe nécessiterait d'utiliser la forme de Backus-Naur (voir https://fr.wikipedia.org/wiki/Forme_de_Backus-Naur par exemple), ce qui n'est pas vraiment prévu dans ce cours.

On remarque qu'il existe en général une forme simple, sans arguments (1), une forme avec des arguments (2)(3) et qu'on modifie le comportement de la commande (on le complexifie) en ajoutant une ou plusieurs options, le "-" précédant chaque option ou chaque groupe d'options.

Dans certains cas les options peuvent avoir elles aussi des arguments. Par exemple, la commande utilisée pour imprimer un fichier :

- `cmd>lpr chemin_vers_le_fichier`
imprimera le fichier sur l'imprimante standard.
- `cmd>lpr -P imp2 chemin_vers_le_fichier`
imprimera le fichier sur l'imprimante de nom `imp2`.

10. Noms de fichiers, méta-caractères, et expressions régulières

Un exemple pour commencer (à tester ; évidemment le choix de `/var/log` n'est là 0 que comme exemple) :

`cmd>ls /var/log`

on voit qu'il y a plein de fichiers dont le nom se termine par `log` (par exemple `boot.log`). Comment ne lister que ceux-ci ?

Pour cela, on utilise le méta-caractère "*" :

`cmd>ls /var/log/*.log`

et on ne voit plus qu'eux.

Explication :

17. C'est en général là-dessus que repose la parallélisme en mémoire partagée : si vous utilisez des logiciels de vidéo par exemple, ils sont en général multithreadés, tant le genre de calcul qu'ils font est coûteux. Avec n cœurs, on peut en théorie calculer n fois plus vite qu'avec un seul.

- “*” correspond à n’importe quelle chaîne de caractères (formée de caractères alphabétiques, numériques et autres). La commande

```
cmd>ls /var/log/*.log
```

va lister tous les fichiers qui sont dans `/var/log` dont le nom commence par n’importe quelle chaîne de caractère, mais finit par `.log`.

- Mais ce n’est qu’une particularisation de quelque chose de plus vaste : les *expressions régulières*. Avec les « bons » outils, on peut, par exemple, trouver dans un texte (dans une chaîne de caractères) tous les *mots* qui se terminent par `.log`.

☛ Quelques autres exemples d’utilisation du joker “*” :

- On peut lister tous les fichiers dont le nom commence par exemple par `syslog` :

```
cmd>ls /var/log/syslog*
```

- Dans ce cas précis, on voit que certains fichiers dont le nom commence par `syslog` ont une extension `.gz` (fichiers compressés).

Pour ne lister qu’eux, je peux faire :

```
cmd>ls /var/log/syslog*gz ou dans notre cas cmd>ls /var/log/syslog*.gz.
```

- Le méta-caractère “*” correspond à n’importe quelle chaîne de caractères, de longueur quelconque.
- Le méta-caractère “?”, lui, correspond à un seul caractère, quelconque.

Exemple (à tester) :

```
cmd>ls /var/log/syslog.?.gz
```

ou bien

```
cmd>ls /var/log/syslog???gz
```

dans ce cas, les 3 caractères “matchés” par ??? peuvent être chacun quelconques et différents.

☛ D’autres manières de filtrer :

- Supposons que je veuille filtrer sur un seul caractère, numérique. Je me sers alors d’intervalle de recherche `[0-9]`, puis :

```
cmd>ls /var/log/syslog.[0-9].gz
```

`[0-9]` va *matcher* tous les caractères de l’intervalle, c’est à dire les chiffres de 0 à 9.

- Si je veux filtrer sur les noms qui commencent par `a`, `b` ou `c`, je peux faire :

```
cmd>ls /var/log/[a-c]*
```

- Bien sûr, on peut tout mixer :

```
cmd>ls /var/log/[a-c]*.[0-9].gz
```

et aussi être plus restrictif :

```
cmd>ls /var/log/[a-c]*.[1-3].gz
```

Attention : l’ordre des caractères alphabétiques est `ABC...Zabc...z` et par conséquent `[a-B]` est un intervalle vide, comme le serait `[3-1]`. En revanche `[A-z]` est l’ensemble de tous les caractères alphabétiques¹⁸.

On peut évidemment utiliser cette technique avec à peu près toutes les commandes. Exemples :

```
cmd>rm *.log
```

```
cmd>mv toto.* /tmp19
```

18. C’est vrai tant qu’on se cantonne à l’utilisation des caractères ASCII ; aujourd’hui on peut représenter « tous » les caractères (Unicode).

19. `mv` : déplace, mais aussi renomme.

Ceci n'est qu'une infime introduction à un monde qu'il serait bon de connaître pour traiter des chaînes de caractères : *les expressions régulières*.

Mais comment est-ce que ça marche ? Il faut en dire un peu plus sur le mécanisme du shell. Que se passe-t-il quand on tape une commande ?

Réponse : la commande est interprétée par le shell :

- (1) Ce qu'on tape est une chaîne de caractères (se terminant par le caractère “retour chariot”).
- (2) Le shell (bash ou autre) analyse cette chaîne. Il peut y avoir des erreurs lexicographiques (emploi de caractères pas reconnus) ou syntaxiques.

Essayez par exemple :

```
cmd>&ls
```

- (3) Une fois cette étape passée, shell extrait le premier mot de la ligne, considérant que ce mot se termine au premier caractère “espace”, et considère que c'est une commande. Il cherche alors cette commande (voir plus loin, page 20) ; s'il ne la trouve pas, il vous le dit, sinon il lance la commande en lui passant le reste de la ligne comme *argument*. Exemple :

```
cmd>ls -l /tmp
```

la commande `ls` reçoit “`-l /tmp`”²⁰.

- (4) Mais si la ligne de commande contient une expression régulière (au lieu de simplement `/tmp`), celle-ci est évaluée et c'est la chaîne résultante qui est passée à la commande : par exemple, si dans mon répertoire courant il existe les fichiers `toto.txt` et `toto.pdf`, la commande

```
cmd>ls -l toto.*
```

est évaluée en :

```
cmd>ls -l toto.pdf toto.txt
```

et c'est la chaîne de caractères “`-l toto.pdf toto.txt`” qui est passée à `ls`.

- (5) Là s'arrête le travail du shell proprement dit : il passe la main à la commande `ls`, créant ainsi un processus dont il est le père, et dont il garde le contrôle ; `ls` analyse cette ligne : est elle *bien formée* ? (la syntaxe des options est-elle correcte, etc.) : le processus `ls` va faire ce qu'il a à faire, et finalement renvoyer au shell un message “tout s'est bien passé”, ou bien un message d'erreur, et le shell reprend complètement la main, et attend de vous une autre commande.♦

Derrière tout ça il y a la notion importante, mais hors de l'horizon de ce cours, de *grammaire formelle* et d'*analyse syntaxique*. Le shell est un interpréteur de commande sophistiqué qui permet d'écrire des programmes dans un langage bien défini, qui suit une *grammaire* bien précise.

11. Deux manières d'acquérir des pouvoirs exceptionnels



Comme on l'a vu, les utilisateurs « standard » n'ont pas tous les pouvoirs²¹, mais il est nécessaire ne serait-ce que pour administrer la machine (ce qui inclut par exemple les mises à jour) d'acquérir des pouvoirs exceptionnels.

Les distributions Linux actuelles proposent deux méthodes :

20. Notons que les espaces en trop sont supprimés : plusieurs espaces successifs se comportent comme un seul.

21. Entre autres, ils n'ont pas accès à certains fichiers ou répertoires, ou bien des accès restreints (lecture seule par exemple).

— *Le superutilisateur* : C’est un utilisateur de nom **root**. Il a, par définition, tous les droits. Donc, il est dangereux d’être **root** ²².

— *La commande **sudo*** : elle permet d’exécuter une commande en étant **root**, le temps que dure la commande :

```
cmd>sudo rm quelque-chose
par exemple.
```

L’idée, avec **sudo** est de ne donner que temporairement les privilèges de **root**, en contrôlant le plus possible ce qui est fait, car :

— Il faut avoir le droit de faire “**sudo**” ; pour cela il faut être membre du *groupe sudo* :

```
cmd>grep sudo /etc/group
pour voir ça.
```

— Il faut donner son propre mot de passe.

— Le mot de passe vous donne le bien droit de faire “**sudo**”, mais ce droit est limité dans le temps (il faut redonner le mot de passe au bout d’un certain temps).

Regardez ce que donnent :

```
cmd>whoami
```

et

```
cmd>sudo whoami
```

Évidemment, la sécurité de “**sudo**” est assez illusoire.

L’utilisateur “**root**” est toujours présent : les utilisateurs d’Ubuntu –par exemple– pourront le vérifier en regardant le fichier **/etc/shadow** ²³ où un “!” désactive la possibilité de se connecter.

En fait, c’est assez dépendant des distributions Linux. Sous les Debian, **sudo** n’est pas installé par défaut, et **root** est un utilisateur comme les autres, avec un mot de passe. Sous Ubuntu on installe **sudo** par défaut ²⁴, mais il suffit de donner un mot de passe ²⁵ à l’utilisateur **root** pour pouvoir aussi se connecter en tant que **root** (depuis l’écran de connexion par exemple).

Notez enfin que **cmd>sudo -i**, c’est exactement comme s’être connecté en tant que **root**.

Quelques avantages de **sudo** :

— On contrôle (via le groupe **sudo**) qui peut acquérir les privilèges de **root**.

— Il faut déjà être connecté pour pouvoir faire **sudo**.

— On peut configurer assez finement ce qui est autorisé pour les utilisateurs qui peuvent faire **sudo** (voir les fichiers et répertoires dont le nom commence par **sudo** dans **/etc**).

Comment devenir **root** quand la machine n’utilise pas **sudo**? On peut évidemment se déconnecter et se reconnecter **root** qui est un utilisateur presque comme les autres, ou bien sans se déconnecter, utiliser la commande **su** :

```
man su
```

```
run a command with substitute user and group ID
```

Dans un terminal, on tape :

```
— cmd>su -
```

Il faut donner le mot de passe de **root**. Le “-” permet de tout faire comme dans un vrai login (initialiser le shell (cf. page 19).

22. ne jamais être **root** en état d’ivresse.

23. **cmd>sudo grep root /etc/shadow**, bien sûr !

24. **cmd>apt install sudo** doit l’installer ; il faudra ensuite ajouter le ou les utilisateurs ad hoc au groupe **sudo**.

25. Question : comment faire ? La commande pour changer *votre* mot de passe est **cmd>passwd**. Pour changer ou définir celui de **root**, il faut... être **root**.

— `cmd>su - commande`

comme pour `sudo` (mais `su` ne se souvient pas de votre mot de passe).



Encore une fois : ATTENTION !

12. Le shell : configuration, environnement

(1) Dans le home directory, quelques fichiers cachés²⁶ sont importants.

Commande	Résultat (chez moi)
<code>cmd>ls ~/.*bash*</code>	<code>/home/moi/.bash_history</code> <code>/home/moi/.bashrc</code> <code>/home/moi/.bash_logout</code>

Et il existe aussi, toujours dans le home directory, un fichier de nom `.profile`

(2) Dans `/etc`

Commande	Résultat (chez moi)
<code>cmd>ls /etc/bash*</code>	<code>/etc/bash.bashrc</code> <code>/etc/bash_completion</code> <code>/etc/bash_completion.d :</code> <code>apport_completion</code> <code>git-prompt gmic</code>

et

Commande	Résultat (chez moi –abrégé–)
<code>cmd>ls /etc/profi*</code>	<code>/etc/profile</code> <code>/etc/profile.d :</code> <code>01-locale-fix.sh gawk.sh</code> <code>vte-2.91.sh</code> <code>bash_completion.sh</code>

À quoi tout cela sert-il ?

- À définir des *variables d'environnement*.
- À définir ou à modifier des commandes.
- Ce qui a un nom contenant **completion** sert à définir les règles de complétion automatique (quand on tape un TAB).

Les variables d'environnement sont globales et peuvent être utilisées par les programmes pour leur permettre d'adapter leur comportement. Un exemple typique : la langue utilisée.

- les fichiers **profile** sont indépendants du shell utilisé.
- L'ordre de parcours est :
 - d'abord le(s) fichiers **profile** de `/etc`,
 - puis `/etc/bash.bashrc`,
 - puis le fichier `~.profile`
 - puis `~.bashrc`.

On peut donc, dans les fichiers `~.profile` et `~.bashrc` compléter ce qui est fait dans `/etc` ou redéfinir des variables.

²⁶. Les fichiers et les répertoires cachés ont un nom qui commence par “.”. La commande `cmd>ls` ne les montre pas, il faut ajouter l'option `a` : `cmd>ls -a` pour les voir.

Qu'est-ce qui est défini ? La commande

```
cmd>printenv
```

permet de voir tout ça. Il y a beaucoup de variables définies !

Remarquons entre autres :

- `SHELL=/bin/bash`

Ici, la variable est `SHELL` et elle contient `/bin/bash`. C'est le shell qu'on utilise bien sûr.

- Des variables dont le nom commence par `LC` : définissent la langue utilisée et les jeux de caractères utilisés.

- etc.

On affecte des valeurs dans le shell en faisant par exemple :

```
cmd>x="coucou"
```

mais pour voir ce que contient la variable, il faut faire :

```
cmd>echo $x
```

Une variable importante : le path. Essayez :

```
cmd>echo $PATH
```

ou

```
cmd>printenv | grep PATH
```

On obtient une liste de chemins dans le système de fichiers, ou les chemins sont séparés par `:`.

Exemple (chez moi) :

```
/home/moi/.local/bin:/usr/local/sbin:/usr/local/bin:
```

```
/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin
```

- Quand on tape une commande, le shell va la chercher successivement dans chacun des chemins de `PATH`, dans l'ordre, et exécute la première trouvée.

- On peut modifier le `PATH`. Pour rajouter un chemin, on peut faire (ici je rajoute `/opt/`) :

```
cmd>PATH=/opt/:$PATH (ajout au début)
```

ou :

```
cmd>PATH=$PATH:/opt/ (ajout à la fin).
```

```
cmd>echo $PATH pour voir le résultat.
```

C'est typiquement le genre de commande qu'on peut rajouter dans `~/.bashrc`.

13. Quelques fichiers et répertoires cachés importants

Certains logiciels créent des répertoires et des fichiers cachés précieux. Attention à ne pas les effacer ! Entre autres :

- `.mozilla/` : répertoire de firefox. Toute votre vie avec firefox !
- `.mozilla-thunderbird/` : le répertoire de thunderbird.
- `.ssh/` si vous utilisez ssh.
- `.VirtualBox/` si vous utilisez Virtualbox.

Le répertoire `.config/` : contient dans des répertoires les configurations de la plupart des logiciels que vous installez. Si vous configurez des logiciels avec les menus (Préférences), vos modifications vont là-dedans. Attention, donc ! Mais il peut être utile d'effacer un répertoire pour « repartir à zéro » avec une application.

14. Une liste de commandes Unix

Commande	Objet	Exemple	Options importantes, remarques
diff	Différences entre fichiers	<code>cmd>diff a b</code>	utiliser entre fichiers texte
file	Devine le type de fichier	<code>cmd>file *.pdf</code>	
mv	Déplace, renomme (voir plus bas)		
sort	Trier un fichier	<code>cmd>sort fich</code>	ajouter <code>-n</code> pour un tri numérique
touch	modifie la date ou crée un fichier vide	<code>cmd>touch toto</code>	
chmod	change les droits (voir plus bas)		
which	trouver une commande dans le PATH	<code>cmd>which uname</code>	
whoami	votre login		
uptime	temps d'activité (depuis le reboot)		
date	date et heure		plein d'options (man date)
df	espace disque utilisé et libre		<code>cmd>df -h</code> , sortie « humaine »

Quelques détails :

- **mv** : déplacer ou renommer. C'est assimilé à une écriture, et donc il faut avoir le droit d'*écrire* sur l'objet qui peut être un fichier ou un répertoire.

Exemples :

- `cmd>mv toto /tmp` déplace le fichier ou le répertoire **toto** dans **/tmp** (il faut aussi avoir le droit d'écrire dans le répertoire but (ici, avec **/tmp**, c'est le cas).
- `cmd>mv toto /tmp/tutu`, on déplace **toto** dans **/tmp** où il s'appellera **tutu**.
- `cmd>mv toto machin`. Le déplacement se réduit à un changement de nom.
- **chmod** : change les droits d'un fichier ou d'un répertoire. Il y a plusieurs façons de procéder, une assez facile, l'autre moins !
- Pour changer les droits du propriétaire, l'option est **u**, c'est **g** pour celle du groupe et **o** pour le reste du monde. Quelques exemples :

- (1) `cmd>chmod u+w toto`,
- (2) `cmd>chmod u+x g+x o-w toto`,
- (3) On peut « factoriser » les ordres : `cmd>chmod ug+x o-w toto` par exemple.
- (4) La méthode « dure » : on considère les 9 digits possibles **xxxxyyyzzz** ; on met un 1 quand on veut que le droit soit ouvert, 0 sinon ; ainsi, pour obtenir **rw-r---**, ceci correspond à : 110 100 000. Bien maintenant, il s'agit de 3 nombres binaires ; il faut les calculer en base 10 :
 - (a) $110 = 1 \times 4 + 1 \times 2 + 0 = 6$,
 - (b) $100 = 1 \times 4 + 0 \times 2 + 0 = 4$
 - (c) $000 = 0$.

Et la commande est `cmd>chmod 640 toto`.

Amusant, non ? (excellent exercice de calcul mental).

15. Bibliothèques (libraries)

Une bibliothèque est un ensemble de pièces de programmes qui peuvent être réutilisées dans un programme. L'intérêt est qu'elles fournissent des fonctions, des structures qui peuvent être utilisées par différents programmes (voir la figure 1 et, par exemple, la référence [18]).

Exemple : la bibliothèque **libc** (plus exactement **glibc**) fournit un grand nombre de fonctions plutôt basiques ; une fonction comme **time** qui permet de récupérer dans un programme l'heure, telle qu'elle est connue par la machine. Un comprend donc l'intérêt à avoir une seule version de cette méthode.

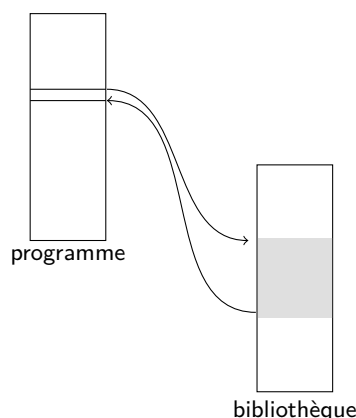


FIGURE 1. Programme et bibliothèque

15.1. Bibliothèques statiques et bibliothèques dynamiques.

- Avec les bibliothèques statiques, les fonctions dont un programme a besoin sont prises dans les bibliothèques au moment de la construction du programme (édition des liens). Le fichier du programme (exemple `/usr/bin/firefox`) contient à peu près tout ce qu'il faut pour faire tourner le programme.
- avec les bibliothèques dynamiques, on va charger en mémoire des fonctions dont on a besoin au moment de leur utilisation.

Quelles sont les bibliothèques dynamiques utilisées par un programme ? La commande est `ldd`. Exemples :

```
— cmd>ldd /usr/bin/firefox. Réponse :
    n'est pas un exécutable dynamique. Donc on utilise des bibliothèques statiques.
— cmd>ldd /bin/ls
    linux-vdso.so.1 (0x00007fff9af8b000)
    libtinfo.so.6 => /lib/x86_64-linux-gnu/libtinfo.so.6 (0x00007ff135523000)
    libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007ff13551d000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007ff135333000)
    /lib64/ld-linux-x86-64.so.2 (0x00007ff1356b9000)
moi@kepler:/2/home/moi/CoursLinux/Cours/Cours4$ man ldd
moi@kepler:/2/home/moi/CoursLinux/Cours/Cours4$ ldd /bin/ls
    linux-vdso.so.1 (0x00007ffd3351f000)
    libselinux.so.1 => /lib/x86_64-linux-gnu/libselinux.so.1 (0x00007fdeb6035000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fdeb5e4b000)
    libpcre2-8.so.0 => /lib/x86_64-linux-gnu/libpcre2-8.so.0 (0x00007fdeb5dbb000)
    libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007fdeb5db5000)
    /lib64/ld-linux-x86-64.so.2 (0x00007fdeb60c0000)
    libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007fdeb5d93000)
```

On voit entre autres, que l'on utilise la `libc` (`libc.so.6`), ce qui est normal, car cette bibliothèque contient beaucoup d'utilités de base.

Les intérêts des bibliothèques sont nombreux : ne pas redévelopper ce qui existe, et utiliser des "services" partagés entre le plus grand nombre de programmes améliore la fiabilité de ces services.

On va voir d'autres bibliothèques quand on va étudier les interfaces graphiques.

Un autre exemple : résoudre le système d'équations linéaires ci-contre.

On peut évidemment coder ça à la main, mais le mieux est d'utiliser une bibliothèque libre (en pratique : `lapack`) : plus de 30 ans d'expérience, fiabilité et performances imbattables. Intéressé(e) par cet exemple ? voir en appendice page 59.

$$2x + y + 3z = 10$$

$$x + y + z = 6$$

$$x + 3y + 2z = 13$$

Nommage des bibliothèques dynamiques. Pour la `libc` qui est un bon exemple, on voit : `/usr/lib32/libc.so.6`

- L'extension `so` est celle des bibliothèques dynamiques.
- Le `.6` est un numéro de version.

16. Interface graphique, bureau etc.

Comment fonctionnent les systèmes de bureau, d'interfaces graphiques que nous utilisons ? C'est assez complexe, ne serait-ce que parce qu'on hérite d'un passé assez lointain (les années 80). La complexité de ces outils outrepassa celle du noyau Linux. Et puis depuis les débuts de ces développements l'environnement matériel a beaucoup changé : les GPU n'implantaient pas les fonctionnalités actuelles.

Une remarque préalable : vous pouvez utiliser une machine Linux sans interface graphique. Tapez `CTRL+Alt F1` et les plus anciens se retrouveront face à quelque chose qui leur rappellera le DOS.

Linux offre plusieurs environnements de bureau permettant tous de gérer des fenêtres des menus, des éléments graphiques etc. Citons : Gnome, Xfce, Mate, Ukui, KDE, mais il en existe d'autres.

Schématiquement, un système de bureau comporte au moins 3 couches, chacune incarnée par une ou plusieurs bibliothèques :

- (1) Un niveau *bas* qui sait créer des fenêtres, interagir avec le clavier et la souris du côté de l'utilisateur et, du côté machine avec des dispositifs gérant directement l'affichage. actuellement ce niveau peut être incarné par deux logiciels :
 - (a) X11, *The X-Window system*.
 - (b) Wayland.
- (2) Un niveau intermédiaire, capable de gérer des boutons, des menus déroulants, le couper-coller, etc. C'est le plus souvent la bibliothèque GTK, développée à l'origine pour le logiciel de manipulation d'images GIMP.
- (3) Le niveau supérieur : c'est gnome, Xfce etc, etc. mais en fait c'est un peu plus compliqué : Xfce par exemple (et certainement les autres aussi s'interfacent avec X11 et GTK).

16.1. X11 ou Wayland ? X11 est ancien (le développement a commencé en 1984), lourd et réputé plus lent que des outils équivalents sous Windows et MacOS. D'où l'idée de développer quelque chose de plus léger et de plus moderne. Il y a eu plusieurs propositions et il semble que Wayland l'emporte.

Certaines distributions utilisent maintenant Wayland (Debian), d'autres garde X11 (Ubuntu). Il faut savoir que pas mal d'applications s'interfacent directement avec X11 et que, pour les faire fonctionner sous Wayland, il a fallu développer... un serveur X11 sous Wayland, au moins pour une période transitoire. X11 permet aussi l'affichage déporté (un programme tourne sur une machine, l'affichage est déporté sur une autre) : l'intérêt pour cette fonctionnalité diminue : on a longtemps utilisé des terminaux X, désormais obsolètes, et un outil comme `x2go` permet un bien meilleur affichage à distance (il est basé sur les techniques de compression d'images utilisées pour la télévision numérique).

Regardons un peu le cas Xfce (les autres sont à coup sûr peu différents) :

La commande :

```
cmd>ps aux|grep -i xfce
```

liste plusieurs processus. C'est donc que :

- Xfce n'est pas monolithique.

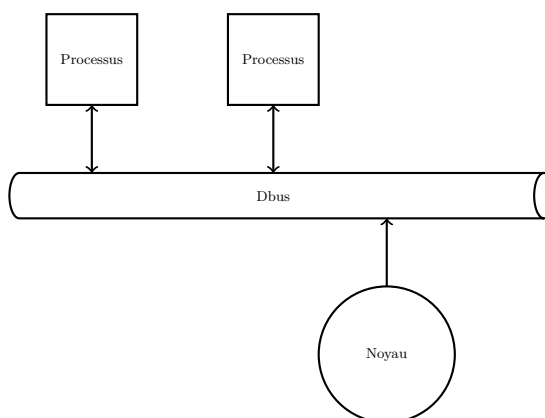


FIGURE 2. Dbus

— Il va obligatoirement que ces processus communiquent entre eux.

Parmi ces processus, on peut regarder **xfce4-session** qui est sûrement important. La commande `cmd>ldd /usr/bin/xfce4-session` liste 82 bibliothèques! Explicitons en quelques unes :

- **libX11** : c'est bien sûr le fameux X11. Notons aussi d'autres bibliothèques qui ensemble forment X11 : **libXrender**, **libXext** etc. (tout ce dont le nom commence par **libX**).
- **libgtk** : définit principalement des « widgets » (boutons, menus etc.).
- **libgdk** : « en dessous » de **gtk**. C'est l'intermédiaire entre **gtk** et le système de fenêtrage (X11 ou Wayland ou...).
- **libpango** : gestion des polices de caractères.
- **libcairo** : dessins bi-dimensionnels, vectoriels. Utilise l'accélération graphique, si c'est possible.
- **libwayland** : la bibliothèque est présente bien que la machine utilise x11.
- **libdbus** : c'est une bibliothèque de communication. On en parle ci-dessous.

Dbus. Dbus est un *bus logiciel*. Citons Wikipédia : « *D-Bus permet à des programmes clients de s'enregistrer auprès de lui, afin d'offrir leurs services aux autres programmes. Il leur permet également de savoir quels services sont disponibles. Les programmes peuvent aussi s'enregistrer afin d'être informés d'événements signalés (parce que gérés) par le noyau, comme le branchement d'un nouveau périphérique* » (voir figure 2).

Il faut mentionner le projet freedesktop.org[23] qui vise à l'interopérabilité des différents environnements de bureau, en définissant des standards et en développant des bibliothèques logicielles (Cairo, par exemple).

17. Les liens

Les liens sont définis sous Unix. Ils permettent d'obtenir une vue d'un fichier ou d'un répertoire « comme s'il était là ». Un lien utilise très peu de ressources.

Sur ma machine, la commande

```
cmd>ls -l /
```

donne (extrait) :

```
lrwxrwxrwx 1 root root 7 avril 27 2020 lib -> usr/lib
lrwxrwxrwx 1 root root 9 avril 27 2020 lib32 -> usr/lib32
lrwxrwxrwx 1 root root 9 avril 27 2020 lib64 -> usr/lib64
lrwxrwxrwx 1 root root 10 avril 27 2020 libx32 -> usr/libx32
```


le `1` en tête indique qu'on a affaire à un lien ; par exemple, `lib` est un *lien symbolique* vers le répertoire `usr/lib`.

On peut vérifier ensuite que les commandes `cmd>ls /lib` et `cmd>ls /usr/lib` donnent le même résultat.

Un lien symbolique :


- S'installe avec la commande `cmd>ln -s`. Exemple :

```
cmd>ln -s /usr/lib lienVersLib
lienVersLib pointe alors vers /usr/lib.
```

- Peut pointer vers n'importe quoi : fichier ou répertoire.

- S'efface avec la commande `rm`. Exemple :

```
cmd>rm lienVersLib.
```

-  Effacer le lien est sans problème (ici `lienVersLib`), mais si on efface l'objet pointé, le lien persiste, mais ne pointe plus sur rien ! Exemple :

```
cmd>touch toto
cmd>ln -s toto tutu (tutu est un lien vers toto).
cmd>rm toto (tutu ne pointe plus sur rien).
cmd>less tutu
```

`tutu: Aucun fichier ou dossier de ce type`

Les liens symboliques sont assez agréables ; ils permettent de créer des sortes d'alias, et puis de contourner un peu la rigidité de l'arborescence de fichiers : un lien peut pratiquement pointer de n'importe quel endroit vers un autre.

Il existe d'autres sortes de liens, les liens « hard », assez peu intéressants en pratique pour les utilisateurs :

- Ils ne peuvent pointer que vers des fichiers.

- Ils équipent le fichier pointé d'un compteur de référence ; ainsi :

```
cmd>ln toto tutu (tutu est un lien « hard » vers toto (notez l'absence de l'option -s).
```

La commande `ls` montre 2 fichiers, alors que les données n'existent qu'une seule fois.

```
cmd>rm toto
```

```
cmd>less tutu : tout se passe bien.
```

18. Systèmes de fichiers

Définition approximative : une organisation hiérarchiques de fichiers et de répertoires.

C'est une définition à peu près indépendante du support matériel de ces systèmes (voir l'article de Wikipédia [22]).

En pratique sous Unix, à chaque fichier ou répertoire est associé un **inode**, qui contient tous les renseignements le concernant. Chaque inode a un numéro, unique. La commande :

```
cmd>ls -li objet
```

permet d'obtenir le numéro d'inode de **objet** (ce qui ne sert pas à grand chose).

Mais les systèmes de fichiers, s'ils montrent au système le même type de structure (arborescence de fichiers et répertoires), ne stockent pas tous en interne les données de la même façon (une bibliothèque est associée à chaque type de système de fichiers), d'où des fonctionnalités et des performances différentes. Citons, pour Linux :

- La famille **ext** : **ext1**, **ext2**, **ext3** et **ext4**. Les premières versions (1 et 2) ont été réalisées par Remy Card, intervenant fréquent aux premières Journées du Logiciel Libre à Lyon [14]. Depuis **ext2**, ce sont les systèmes de fichiers standards sous Linux (actuellement, c'est **ext4**).
- **Jfs** : (IBM ; peu gourmand en ressources).
- **ReiserFS** : semble abandonné.
- **Xfs** : très grande capacité de stockage, et rapide pour les très grands systèmes de fichiers.

- **Btrfs** : selon les auteurs, c'est le *B-Tree file-system* ou le *Better file system*. Génial sur le papier, mais ne semble pas percer pour l'instant (standard cependant chez Suse).
- **Zfs** : (Open-Zfs). Origine SUN, assez proche de **Btrfs** qui semble en dériver, en tout cas conceptuellement.
- Les systèmes Linux peuvent aussi manipuler (et créer) des systèmes de fichiers propriétaires (**ntfs**, **vfat**, **fat** etc.).
- Systèmes de fichiers en réseau : une machine *exporte* une partie de son arborescence vers d'autres machines ; sous Unix (et donc Linux), cela s'appelle **nfs** (network file system) ; **samba** fait ça aussi.
- Il existe aussi des *clustered file-systems* pour la lecture et l'écriture en parallèle (Lustre, BeeGFS, GPFS, Xtremefs, GlusterFS, MooseFS, XrootD, EOS...) ; ça c'est pour Youtube ou le Centre de Calcul des Physiciens Nucléaires à La Doua²⁷.

Quelques détails :


- **Btrfs** et **Zfs** permettent de faire des *snapshots*, soit en français des *instantanés* : obtenir une *vue* du système de fichiers tel qu'il est à un instant donné, en le laissant évoluer par ailleurs. On trouvera en annexe page 61 une description (à coup sûr simplifiée) de la technique utilisée (le COW).
- **Btrfs** et **ReiserFS** utilisent tous les deux une représentation des données sous forme d'*arbre binaire équilibré* (*B-Tree*), décrite aussi en annexe page 62, qui assure une très grande vitesse de lecture.

Un des problèmes majeurs est la fiabilité des systèmes de fichiers : outre qu'un bug pourrait avoir des conséquences désastreuses, il faut absolument tenir compte de l'accident qu'un arrêt brutal de la machine (coupure de courant par exemple) pourrait causer. Le risque est d'avoir un système de fichiers incohérent, car l'interruption aura eu lieu alors qu'une écriture était en cours (apparition d'orphelins par exemple). Regardons pour cela l'histoire des systèmes **ext** [1-4] :

- **ext1** était un prototype ; **ext2** (Rémy Card, 1993 [14]) est rapidement devenu le système standard sous Linux. Il limite la fragmentation du support, n'implante pas de sécurité et a fini par apparaître comme limité quant à la taille et au nombre des objets stockés.
- **ext3** (2001) : le principal apport est la *journalisation* qui permet de *rejouer* une E/S qui aurait été interrompue.
- **ext4** (2008) : la limite de la taille est portée à 2⁶⁰ (!) octets. Une autre avantage est de réduire encore la fragmentation. C'est actuellement le système de fichiers standard sous la quasi totalité des distributions Linux.

La journalisation : On consultera [33] pour une description détaillée. Le principe de la *journalisation* consiste à maintenir une liste (en général dans un fichier) des modifications apportées au système de fichiers, de manière indépendante des modifications elles-mêmes, pour pouvoir rejouer ces modifications en cas de crash, ou au moins de remettre le système dans un état cohérent. C'est ce qui est implanté dans **ext3** (avec différentes options, voir référence ci-dessus).

18.1. Manipulation des systèmes de fichiers (partitionner, créer, monter, etc.)

Tout, ici, doit être fait en tant que **root**, donc  **Attention, danger !**

On ne va parler que des systèmes de fichiers classiques, physiques (**ext4**, **vfat** etc. Pour **btrfs**, voir par exemple[5]).

- (1) Un système de fichiers s'installe toujours dans une *partition*. Même si le périphérique sur lequel vous allez installer le système de fichiers ne va en contenir qu'un, vous devez partitionner (créer au moins une partition).

27. Ou pour le futur télescope à grand champ Vera-C. Rubin : 20 téraoctets par nuit, à conserver au moins 10 ans [12].

- (2) On crée ensuite le système de fichiers en utilisant la commande **mkfs** qui installe dans la partition la structure de donnée nécessaire, prête à recevoir vos fichiers et répertoires.
- (3) Ensuite il faut *monter* le système de fichiers pour qu’il soit incorporé à l’arborescence de la machine (ceci vaut aussi pour les système de fichiers en réseau).

18.1.1. *Créer une (des) partitions(s), formater.* En ligne de commande, la commande est **fdisk**. Un outil graphique, plus simple à utiliser, est **gparted**.

La première chose à faire est de connaître le nom du *device* (périphérique à formater). Répétons : il faut être **root**, donc :

- soit vous êtes connecté en tant que **root** (par **cmd>su** - par exemple),
- soit vous préfacez toutes les commandes ci-dessous par **sudo** si votre machine est amie de **sudo**. Vous pouvez aussi faire une fois pour toutes, dans ce cas : **cmd>sudo -i**.

Dans tous les cas :  !

Pour connaître le périphérique que vous voulez formater vous pouvez par exemple, dans le cas d’un périphérique usb, regarder ce que donne la commande :

```
cmd>dmesg
```

après avoir introduit le périphérique. Vous verrez par exemple quelque chose comme :

```
[sdd] Attached SCSI removable disk.
```

Vous pouvez aussi installer et utiliser la commande **ls SCSI**, bien pratique. Il faut ensuite taper une commande comme **df** pour voir si le périphérique est monté. Si oui, le démonter (voir plus bas).

On lance la commande

```
cmd>fdisk /dev/sdd (si votre périphérique est bien /dev/sdd). Ensuite, la commande est interactive ; il faut :
```

- (1) Effacer les partitions existantes (**p** pour les lister, **d** pour en effacer une).
- (2) **n** pour créer une partition (il y en a différents types, mais utilisons celle par défaut pour linux). On crée une partition primaire, et on en donne la taille (par défaut on utilise tout le périphérique).
- (3) On peut comme ça créer plusieurs partitions, qui vont s’appeler **/dev/sdd1**, **/dev/sdd2** etc (parce que mon périphérique est **/dev/sdd**).
- (4) **w** pour écrire les résultats ; En fait jusque là, on n’a rien changé sur le disque. En sautant cette étape, on peut encore tout sauver !
- (5) **q** pour quitter.

18.1.2. *Installer un système de fichiers (formater).* C’est très simple. Mes partitions s’appellent par exemple **/dev/sdd1**, **/dev/sdd2** etc. Pour installer un système de fichiers **ext4**, il suffit de faire :

```
cmd>mkfs -t ext4 /dev/sdd1 (et bien sûr remplacer 1 par ce qu’il faut le cas échéant...).
```

18.1.3. *Monter le système de fichiers.* Monter un système de fichiers consiste à *accrocher* un système de fichiers à un répertoire existant dans l’arborescence. On peut utiliser un répertoire *existant* (attention, je vais cacher ce qui est éventuellement dedans), ou en créer un. Supposons que j’utilise **/mnt** comme point de montage. La commande est alors :

```
cmd>mount -t ext4 /dev/sdd1 /mnt
```

Il y a 3 arguments :

- (1) le type de système de fichiers, ici : **-t ext4**,
- (2) ce qu’on monte, ici : **/dev/sdd1**,
- (3) où on le monte, ici : **/mnt**.

Et voilà.

Dans notre cas, **/mnt** est accessible. Ensuite, rien ne prouve que le répertoire ainsi monté sera accessible à tout le monde : à priori, il va appartenir à **root**. Il faut donc en changer les droits, ou le propriétaire. On va voir une autre façon de faire.

18.1.4. *Démonter le système de fichiers.* Il faut utiliser `umount`. Dans l'exemple ci-dessus on peut soit faire :

```
cmd>umount /mnt
soit :
cmd>umount /dev/sdd1
```

18.2. Le fichier `/etc/fstab`. Si on souhaite automatiser le « mount » de systèmes de fichiers, il faut utiliser le fichier `/etc/fstab`, et donc le modifier²⁸.

`/etc/fstab` contient la description de tout ce qui doit ou peut être monté, en précisant quoi doit être monté et où. L'utilisation principale, mais pas exclusive, consiste à indiquer au système ce qu'il doit monter au boot.

18.2.1. *`fstab`, à l'ancienne.* Exemple :

# <file system>	<mount point>	<type>	<options>	<dump>	<pass>
/dev/sda2	/	ext4	errors=remount-ro	0	1
/dev/sda1	/boot/efi	vfat	umask=0077	0	1

Le premier champ est ce qu'on monte, le deuxième où on le monte et le troisième le type de système de fichiers. On voit ainsi que la partition `/dev/sda1`, formatée en `vfat` sert pour le boot (c'est `uefi`). Passons sur le champ "dump", le dernier champ dit l'ordre dans lequel les systèmes de fichiers sont vérifiés au boot. Les options sont nombreuses, ici :

- `errors=remount-ro` dit de remonter sans droits d'écriture en cas d'erreur,
- `umask=0077` indique les droits qu'on enlève : ainsi 077 laisse tous les droits `rwX` au propriétaire (root), et enlève tous les droits (car 7 en binaire est 111) au groupe et aux "autres". Les droits de `/boot/efi` sont donc `rwX-----`.

18.2.2. *`fstab`, plus moderne.* Citer les noms des partitions comme `/dev/sda1` est source d'erreur. Le système propose un identificateur unique pour chaque partition, qu'on peut obtenir avec la commande `blkid`. Chez moi, cela donne :

```
cmd>blkid /dev/sda1
/dev/sda1: UUID="4454-89BE" BLOCK_SIZE="512" TYPE="vfat" PARTLABEL="EFI System Partition"
PARTUUID="ad32b00b-6810-45f6-93d2-d946eb611524"
cmd>blkid /dev/sda2
/dev/sda2: UUID="c447c9ae-2491-4526-8fd8-8a56f79b2e0b" BLOCK_SIZE="4096" TYPE="ext4" PARTUUID="2eb75d027-4383-b811-48f999adf3a2"
```

on récupère les UID... et le fichier `/etc/fstab` devient :

# <file system>	<mount point>	<type>	<options>	<dump>	<pass>
UUID="c447c9ae-2491-4526-8fd8-8a56f79b2e0b"	/	ext4	errors=remount-ro	0	1
UUID="4454-89BE"	/boot/efi	vfat	umask=0077	0	1

18.2.3. *Autoriser un utilisateur à monter un système de fichiers.* Dans notre cas (monter `/dev/sdd1` sur `/mnt`), la ligne à ajouter à `/etc/fstab` est :

```
/dev/sdd1 /mnt ext4 noauto,user Ici :
```

- `user` donne le droit de monter le système de fichiers aux utilisateurs (et donc aussi de démonter).
- `noauto` : le système de fichiers ne sera pas monté au moment du boot.

On peut aussi récupérer l'UUID et écrire quelque chose comme :

```
UUID="6e9d5173-3e30-4e8b-8546-b82a7a311c57" /mnt ext4 noauto,user
```

N'importe quel utilisateur peut alors taper :

28. Il faut bien sûr avoir les pouvoirs de `root`. Ensuite, toucher à ce fichier est dangereux. Il faut :

- (1) Le copier pour en garder la version actuelle : `cmd>cp /etc/fstab /etc/fstab.save`
C'est une recommandation qui vaut pour tous les fichiers « sensibles ».
- (2) Utiliser un éditeur de texte pour le modifier. Vous en avez plusieurs à disposition : `vim` (pour les masochistes), `gedit` ou `nano`, voir même `emacs`. Pour ce genre de chose, je recommande `nano`.

```
cmd>mount /mnt
et
cmd>umount /mnt
```

19. Une application complète : sauvegarde d'un répertoire avec rsync

But. : réaliser une sauvegarde « intelligente » d'un répertoire vers un autre (à priori sur un support externe, connecté en usb).

Rsync. La commande rsync est très puissante²⁹ ; à priori c'est une simple copie d'un répertoire vers un autre, par exemple :

```
cmd>rsync -a /home/ /mnt/home/
```

va copier le répertoire `/home/` vers `/mnt/home/`. Mais attention, *on ne va copier que ce qui n'est pas déjà présent dans /mnt/home/, ou ce qui est déjà présent dans /mnt/home/ mais a été modifié dans /home/*. Le but est de synchroniser (d'où le nom de la commande) les deux répertoires en faisant le minimum de travail. L'option `-a` est nécessaire (on conserve les droits, les propriétaires, les groupes des objets). Attention, les chemins (ici `/home/` et `/mnt/home/` doivent avoir un `/` à la fin.

— On peut aussi ajouter `-delete` (oui, il y a bien deux “-”) :

```
cmd>rsync -a -delete /home/ /mnt/home/
```

alors, si des fichiers existent dans `/mnt/home/` sans exister dans `/home/`, ils sont effacés dans `/mnt/home/`.

Avec ces deux options, on est donc capable de faire une sauvegarde parfaite³⁰.

Autres options de `rsync`, souvent utiles :

— `Verbose` : `-v` : vous dit tout ce que fait rsync. C'est rapidement beaucoup trop bavard, mais c'est très utile pour la mise au point !

— `-dry-run` : ne transfère rien, mais, associé à `-v` permet de voir ce qui se passerait sans l'option.

Avant toute copie, ou la première fois il est vraiment recommandé de faire un test à blanc :

```
cmd>rsync -av -delete -dry-run /home/ /mnt/home/
```

Bien regarder ce qui se passe !³¹

19.1. La sauvegarde. On va d'abord faire les choses « à la main », et puis on automatisera la sauvegarde.

Que faut il faire ?

- (1) Monter la partition sur laquelle on va faire la sauvegarde, si c'est nécessaire.
- (2) Avec `rsync` synchroniser les deux répertoires : celui à sauvegarder et la sauvegarde.
- (3) Éventuellement, démonter la partition de sauvegarde.

Ne pas garder la partition de sauvegarde montée en permanence diminue les risques d'écriture intempestifs³²

Supposons que ma partition de sauvegarde soit toujours `/dev/sdd1`, qu'on la monte sur `/mnt` et qu'on veuille sauvegarder `/home`. Alors, les trois étapes ci-dessus sont :

29. `rsync` n'est pas forcément installé par défaut :

```
cmd>sudo apt install rsync
```

ou l'équivalent pour l'installer.

30. Le contenu de `/mnt/home/` sera bien identique à celui de `/home/` après.

31. Il peut être judicieux de faire : `cmd>rsync -av -delete -dry-run /home/ /mnt/home/ | less`

32. Et aussi, certains disques externes s'éteignent une fois démontés.

(1) `cmd>mount /mnt`³³

(2) Synchroniser avec `rsync`. La commande est :

```
cmd>rsync -a -delete /home/ /mnt/home/
```

Options à considérer, en tout cas pour la mise au point :

— `-v` : *verbose*, bavard.

— `-dry-run` : ne rien faire, ne rien copier, ne rien effacer.

Donc, pour la mise au point, on peut vérifier que tout se passe bien avec :

```
cmd>rsync -av -dry-run -delete /home/ /mnt/home/
```

³⁴


Ne pas oublier les “/” À LA FIN pour les répertoires : `/home/`, `/mnt/home/` (et non pas `/home` ni `/mnt/home`).



(3) Éventuellement, démonter la partition de sauvegarde :

```
cmd>umount /mnt
```

19.2. Automatiser la sauvegarde. Le but est de faire une sauvegarde qui se déclenche automatiquement tous les jours.

19.2.1. *Première étape : faire un « script ».* Un script, c’est un programme pour le shell.

Pour cela on va créer un fichier qui contient les commandes vues précédemment (après avoir tout testé à la main!). Voici le fichier, qu’on peut appeler par exemple `sauv.sh`.

```
#!/bin/bash
mount /mnt
rsync -a --delete /home/ /mnt/home/ &>>/var/log/sauv.log
umount /mnt
```

Commentaires :

- La première ligne est là là pour dire que le script doit être exécuté avec le shell *bash*.
- `&>>/var/log/sauv.log`. On se souvient de la redirection des entrées-sorties ; En fait il y a plusieurs sorties numérotées 1 (normale), 2 (erreur). Le `&` redirige toutes les sorties ; on les redirige vers `/var/log/sauv.log` et les deux “>” (») rallongent le fichier `/var/log/sauv.log` (on n’écrase pas le contenu de `/var/log/sauv.log`, ce qui serait le cas avec un seul >).

Où installer ce fichier ? par exemple dans `/etc/cron.daily`. Une fois créé avec l’éditeur de texte, on le rend exécutable (`cmd>chmod +x /etc/cron.daily/sauv.sh`).

Comment l’exécuter automatiquement ? C’est fait ! Il tournera tous les jours où la machine fonctionnera (c’est parce qu’on l’a mis dans `/etc/cron.daily`).

33. Là, je suppose qu’il y a la bonne ligne dans `/etc/fstab`. Sinon, on peut toujours faire :

```
cmd>mount -t ext4 /dev/sdd1 /mnt
```

tout aussi efficace.

34. Il n’y a aucun problème à interrompre `rsync` avec `Control+C`

Il reste une chose à faire : le fichier `/var/log/sauv.log` va grossir indéfiniment. Il faut le faire « tourner » avec `logrotate`.

On peut regarder les différents scripts présents dans `/etc/logrotate.d` et en déduire que celui ci devrait convenir (ce n'est pas très critique ; `cmd>man logrotate` en dira plus sur les options du fichier) :

```
/var/log/sauv.log
{
rotate 4
weekly
missingok
notifempty
compress
delaycompress
sharedscripts
endscript
}
```

Installer ce fichier sous le nom `sauve` par exemple dans `/etc/logrotate.d/`.

19.3. Sauvegarder plusieurs répertoires. On peut assez facilement transformer le *script* ci-dessus page 30 pour sauvegarder plusieurs répertoires. Il peut être intéressant de sauvegarder `/home`, mais aussi `/etc` par exemple, qui contient toute la configuration de la machine. Pour cela, on va un peu programmer le shell (car le shell est un langage de programmation).

19.3.1. *Un premier script.* Fabriquons³⁵ un fichier de nom `essai1.sh` (le nom n'a pas d'importance) :

```
#!/bin/bash
for d in /home/ /etc/;do
echo $d
done
```

Nous avons déjà rencontré la première ligne. À la deuxième ligne nous commençons une boucle ; la variable `d`³⁶ va contenir successivement les chaînes de caractères `/home/` et `/etc/`. Tout de qui est compris entre la ligne `for...; do` et la ligne `done` est effectué successivement avec les valeurs `/home/` et `/etc/` stockés dans la variable `d`. À la troisième ligne, on écrit le contenu de `d` (le shell fait une différence entre une variable (`d` ici) et son contenu qui ici est `$d`). Enfin la dernière ligne termine la boucle `for`. Une fois le fichier écrit, rendons le exécutable :

`cmd>chmod +x essai.sh`

et lançons la commande :

`cmd>./essai.sh`. On voit s'imprimer successivement `/home/` et `/etc`.

19.3.2. *Deuxième étape.* On réécrit le script vu page 30 :

35. Il faut utiliser un éditeur de texte, genre `nano` ou autre.

36. On aurait pu choisir un nom complètement différent.

```
#!/bin/bash
mount /mnt
for d in /home/ /etc/;do
rsync -a --delete $d /mnt/$d &>>/var/log/sauv.log
done
umount /mnt
```

Et voilà : `d` contiendra succesivement `/home/` et `/etc/`, `/mnt/$d` vaudra succesivement `/mnt/home/` et `/mnt/etc/`, ce qui fera bien ce qu'on cherche³⁷. Notons que `/etc/` ne subissant que peu de modifications chaque jour, il n'y aura pratiquement aucun travail à effectuer pour le sauvegarder, *après* la première sauvegarde.

19.4. Synchronisation à distance. `Rsync` permet des synchronisations de répertoires entre machines distantes, à condition qu'on puisse établir une connexion cryptée (`ssh`) entre les machines.

Supposons que je dispose d'un compte utilisateur `untel` sur la machine distante `mm.xxx.yy`. Je synchroniserai le répertoire distant `/sauv/` (par exemple) avec le répertoire `/home/` de ma machine locale avec la commande :

```
cmd>rsync -a -delete -e ssh /home/ untel@mm.xxx.yy:/sauv/
```

il peut être bien utile d'ajouter l'option `-z` pour compresser les données transférées et limiter la bande passante sur le réseau :

```
cmd>rsync -az -delete -e ssh /home/ untel@mm.xxx.yy:/sauv/
```

Une telle sauvegarde est absolument sûre, car la liaison est cryptée³⁸

Ce qui suit n'est pas propre à Linux, ni à Unix, bien sûr. La deuxième partie (commandes Linux, page 39), elle, est spécifique à Unix (Linux).

20. Réseau : le modèle en couches de l'Open Systems Interconnection (OSI)

L'idée est comme souvent en informatique de cacher de l'information : la couche de niveau n du réseau n'a pas à savoir comment fonctionne la couche de niveau $n - 1$, comment elle assure son service (voir [29]). Ce qui va nous intéresser apparaît aux niveaux 3 et 4 ; les niveaux 1 et 2 sont chargés des aspects matériels du réseau. Notez que, dans ce modèle, on peut changer la réalisation pratique d'un niveau sans que les niveaux supérieurs en soient affectés.

Ce qui va nous intéresser :

- Le niveau 3 : Internet Protocol, avec deux versions, IPv4 et IPv6. Ce niveau propage des paquets d'information, de taille bornée.
- Le niveau 4. Les paquets du niveau 3 transportent les données utiles. Les données transportées sont d'un type donné, spécialement :
 - ICMP (Internet Control Message Protocol) : transmission de messages à usage interne du réseau (messages d'erreur, echo, etc.)[25].
 - UDP et TCP : UDP[35] est un protocole *léger* : peu de contrôles, pas d'accusé de réception à la différence de TCP[34].

37. `rsync` crée automatiquement le répertoire de destination s'il n'existe pas.

38. Sur la machine `mm.xxx.yy`, on peut aussi faire en sorte que le répertoire `/sauv/` soit crypté, mais ce n'est pas du ressort de `rsync`.

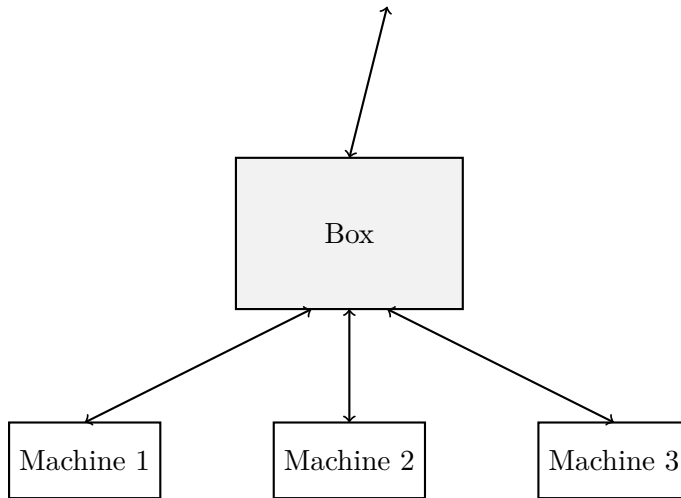


FIGURE 3. machines derrière une « box »

A ce niveau, on définit aussi la notion de *port*, étiquetés par des numéros et associés à des services ou des applications.

Les paquets de données sont transportés sans connexion, c'est à dire que les données qu'ils contiennent sont suffisantes pour assurer leur livraison, sans qu'une connexion figée soit établie ; on parle de *datagramme*[21]. Un des inventeurs de la notion de datagramme est Louis Pouzin[27]³⁹.

21. Réseau, configuration classique : machines derrière une « box »

21.1. Le cas « classique » : IPv4. Toute machine a une adresse, qui en IPv4 ressemble à 134.214.100.121 et chaque adresse est unique sur l'internet. Les 4 nombres qui forment l'adresse sont stockés chacun sur un octet ce qui correspond à des nombres compris entre 0 et 255 (bornes incluses)⁴⁰. L'adresse IPv4 est donc stockée sur 4 octets (soit 32 bits) et le nombre d'adresses disponibles est donc de $2^{32} - 1 = 4294967295$ soit un peu plus de 4 milliards d'adresses. Sachant que, comme on va le voir, certaines adresses sont réservées à des usages particuliers, c'est peu et actuellement insuffisant. IPv4 peut donc être considéré comme un protocole en fin de vie.

La technique utilisée derrière les « box » est celle du *réseau local* (voir figure 3) : seule la « box » est visible du monde extérieur, possède une adresse ip connue de l'internet. C'est ce qui a permis à IPv4 de survivre jusqu'à aujourd'hui, car cela permet de cacher des machines à l'internet. La « box » a des interfaces sur deux réseaux : le réseau local, et le réseau extérieur.

21.1.1. Le réseau local :

- les machines du sous réseau local : elles ont des adresses dont les deux octets de gauche sont 192.168. Ces adresses ne sont certainement pas uniques, mais sont invisibles de l'internet. La « box » agit comme un barrage, laissant quand même passer (on va voir comment) ce qui est nécessaire.

39. Intervenant aux JDLL il y a quelques années.

40. Car $255 = 1 + 2 + 4 + \dots + 2^7 = 2^8 - 1$.

En-tête IPv4																																																							
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31																								
Version d'IP				Longueur de l'en-tête				Type de service								Longueur totale																																							
Identification																Indicateur		Fragment offset																																					
Durée de vie								Protocole								Somme de contrôle de l'en-tête																																							
Adresse source																																																							
Adresse destination																																																							
Option(s) + remplissage																																																							

FIGURE 4. En-tête IPv4

Cela pose un certain nombre de questions :

- (1) Comment sont affectées les adresses aux machines du réseau local ? à ma « box » ?
- (2) Comment les machines du réseau local communiquent-elles avec le mode extérieur ?
- (3) Comment fait-on, depuis une machine du réseau local, pour joindre une machine extérieure au réseau local, connaissant son nom ?

Et bien sûr quels sont les outils Linux (Unix) associés, quelles sont les commandes qui permettent d'explorer ce fonctionnement ?

Auparavant il faut comprendre un peu comment fonctionne IP (v4 dans ce cas).

21.2. Internet protocol. Un paquet IP est formé d'une en-tête et de données. L'en-tête pour IPv4 est décrite à la figure 4. Pour une description détaillée, consulter par exemple [9]. Parmi les champs de l'en-tête, retenons :

- L'adresse source : adresse IP de l'émetteur.
- L'adresse du destinataire.
- Le protocole (8 bits) : numéro du protocole au-dessus de la couche réseau : TCP = 6, UDP = 17, ICMP = 1.
- Identification (16 bits) : Numéro permettant d'identifier les fragments d'un même paquet.

21.3. Le NAT (Network Address Translation. Supposons que la machine 1 (figure 3) veuille envoyer un paquet à l'extérieur. La box (un système Linux, à priori) va changer l'adresse ip de la machine 1 par sa propre adresse, et se souvenir d'un identifiant unique du paquet. Au retour, il enverra le paquet de retour vers la machine 1 (c'est un peu simplifié). Donc, de l'extérieur, on ne voit que la « box »⁴¹

Comment tester ça ? Il suffit d'aller sur le site <https://www.whatismyip.com/fr/> : si vous avez plusieurs machines dans votre réseau local (ordinateur, téléphone sur le wi-fi, etc.) quelque soit la machine que vous utiliserez pour consulter cette url, vous obtiendrez la même adresse v4 (à condition bien sûr que vous soyez encore sur un réseau IPv4).

Bien, mais ceci ne nous dit pas :

- (1) Comment chaque machine du réseau local est configurée ? Comment connaît-elle les paramètres nécessaires à son fonctionnement (entre autres son adresse ip) ?

41. Le NAT est une fonctionnalité du noyau Linux. Voir page 41.

- (2) Comment la box sait-elle faire la différence entre un trafic local et un accès vers l'extérieur ?
- (3) Comment associer un nom (exemple : `www.aldil.org`) à une adresse ip et vice versa ?

21.3.1. *La configuration réseau d'une machine locale.* Jadis, on faisait ça à la main en remplissant un fichier. Pas très facile, ni pratique pour un portable appelé à changer de réseau !

C'est un service : une machine dans le réseau (dans notre cas c'est la box, mais ce peut être une autre) fait tourner un démon DHCP. De façon sommaire : la machine en manque de configuration (quand elle boote par exemple) envoie un paquet IP en *broadcast* (à la cantonade !) : dans ce cas l'adresse du destinataire est 255.255.255.255. Le paquet contient entre autres l'adresse MAC (voir ci-dessous) de la machine *cliente*. Le serveur DHCP envoie alors tous les renseignements nécessaires à la configuration réseau de la machine cliente, dont son adresse IP. Il se base sur l'adresse MAC, pour lui associer une adresse IP⁴² ; il envoie aussi tous les autres paramètres dont la machine a ou peut avoir besoin (voir [8] pour une explication un peu détaillée). Les renseignements envoyés par le serveur ont une durée de vie, qui peut être limitée ou infinie : quand les renseignements sont périmés, la machine cliente refait une requête au serveur DHCP.

21.3.2. *l'adresse MAC.* est un identifiant d'une interface réseau : c'est attribué à la construction physique de la machine, et c'est à priori unique. Exemple :

d8:50:e6:ba:2e:d8

21.3.3. *Que faire des paquets qui ne sont pas destinés à une machine locale ?* Il faut les envoyer à une machine qui fait office de porte de sortie (GATEWAY). La machine du réseau local doit donc connaître l'adresse IP de cette porte (ça fait partie des paramètres renvoyés par le serveur DHCP). La machine GATEWAY a deux interfaces réseau configurées, et donc deux adresses IP :

- (1) une sur le réseau local (celle que connaissent les machines du réseau local),
- (2) une adresse sur le réseau extérieur (celui du FAI dans notre cas). C'est celle qu'on voit quand on va sur le site <https://www.whatismyip.com/fr/>.

Dans notre cas, la GATEWAY c'est la box, mais ce n'est pas obligatoire.

21.3.4. *Netmask (masque de sous-réseau).* Supposons que mon adresse ip soit 51.75.22.226. Quelles sont les adresses qui font partie du même sous-réseau que moi ? Le sous-réseau peut être mon réseau local privé derrière ma box, ou, disons le réseau free.fr ou... ?

La règle est que les adresses d'un même sous-réseau partagent un début d'adresse (partie gauche) commune. On trouve en général la notation 192.168.0.10/24. Dans ce cas cela veut dire que les 24 premiers bits en partant de la gauche sont communs aux adresses du réseau. Ici $24 = 3 \times 8$ et donc les adresses de l'ensemble des machines du réseau (local) sont de la forme 192.168.0.*. Le masque donne donc aussi la taille du sous-réseau (254 adresses ici⁴³).

Toujours dans l'exemple ci-dessus, le nombre de 4 octets ayant 24 de ses bits de gauche à 1 et les autres à 0 s'écrit (en base 2 et en séparant les octets par un ".") :

42. Plusieurs méthodes sont possibles : table faisant correspondre adresse MAC et adresse (l'adresse de la machine est alors fixe), ou tirage aléatoire.

43. La valeur 255 est réservée comme on le verra plus bas.

11111111.11111111.11111111.00000000

et comme 11111111 = 255 en base 10, cela peut se noter :

255.255.255.0

qui est le *netmask* du réseau.

Cela permet de répondre à la question suivante : étant donné une adresse IPv4, est-elle dans le même réseau que moi ? Là, il faut faire un peu de logique : on assimile 0 à la valeur *faux* et 1 à la valeur *vrai*⁴⁴. On a :

- 0 ET 0 = 0 (faux et faux est faux)
- 0 ET 1 = 0 (faux et vrai est vrai)
- 1 ET 0 = 0 (vrai et faux est faux)
- 1 ET 1 = 1 (vrai et vrai est vrai).

On fait un « ET » bit à bit entre le netmask et l'adresse testée.

Exemples :

- Considérons l'adresse 192.168.0.21, le netmask étant 255.255.255.0.
192.168.0.21 est 11000000.10101000.00000000.00010101 en binaire.
Le netmask est 11111111.11111111.11111111.00000000.
Le « ET » entre le netmask et mon adresse ip est :

11000000.10101000.00000000.00010101
11111111.11111111.11111111.00000000

11000000.10101000.00000000.00000000

Le résultat, 11000000.10101000.00000000.00000000, est 192.168.0.0 en base 10.

- Pour l'adresse 192.168.0.32, le calcul précédent donne le même résultat : les deux adresses sont donc dans le même réseau.
- Mais pour 134.214.100.8, le résultat est différent, et les deux adresses ne sont donc pas dans le même réseau.

21.3.5. *Adresse de broadcast*. Comment contacter toutes les machines du même réseau ?

On considère la partie commune des adresses du réseau : dans mon cas, utilisant le netmask 255.255.255.0, et mon adresse étant 192.168.0.32, j'en déduis que les machines de mon réseau ont toutes des adresses de la forme 192.168.0.x ou x est compris entre 0 et 255. L'adresse de broadcast est obtenue en remplaçant « x » par 255 (soit des 1 partout en binaire), ce qui ici donne :

192.168.0.255.

Envoyer un paquet IP avec cette adresse de destinataire, c'est écrire à toutes les machines du réseau.

Noter que :

44. Vous pouvez aussi voir ça comme une multiplication, remplacer ET par \times .

- L'adresse 192.168.0.255 est réservée pour le *broadcast* : elle ne peut donc pas être affectée à une machine particulière et, dans le cas envisagé ici, il ne peut y avoir que 254 machines dans notre réseau local.
- On a déjà vu un *broadcast* à propos du DHCP, où l'adresse était 255.255.255.255 pour le *broadcast* . Ce n'est pas exactement la même chose.

21.3.6. *Serveur de noms (DNS, Domain Name System)*. La configuration réseau d'une machine contient au minimum 4 paramètres :

- (1) l'adresse IP,
- (2) le netmask,
- (3) l'adresse de la GATEWAY,
- (4) et l'adresse du *serveur de noms*.

Le *serveur de noms* est chargé d'associer un nom (`www.aldil.org` par exemple) à une adresse ip, et vice-versa.

Supposons que je cherche l'adresse de `math.univ-lyon1.fr` ; le fonctionnement est à peu près le suivant :

- (1) Est-ce que ma machine connaît l'adresse IP de `math.univ-lyon1.fr` ? Si oui, c'est fini. Sinon, ma machine envoie une requête au serveur de noms précédemment défini (par la configuration de ma machine –obtenue par le serveur DHCP de mon réseau–).
- (2) Est ce que ce serveur de noms connaît l'adresse que je cherche ? Dans mon cas le serveur de nom peut être un serveur qui tourne dans la box, ou bien celui de mon FAI, disons `free.fr`. Si le serveur a la réponse, c'est fini, sinon, il faut monter d'un cran.
- (3) Si le serveur de noms de `free.fr` connaît l'adresse du serveur de nom de `univ-lyon1.fr`, il lui demande l'adresse de `math.univ-lyon1.fr` ; peut être faut-il une étape supplémentaire si `univ-lyon1.fr` est inconnu chez `free.fr` : dans ce cas il consulte le DNS de `fr`, qui connaît *obligatoirement* `univ-lyon1.fr`.
- (4) Le serveur de noms de `univ-lyon1.fr` connaît *obligatoirement* l'adresse cherchée (ainsi que les adresses de toutes les machines du domaine `univ-lyon1.fr` : c'est stocké dans des tables).
- (5) Une fois un serveur capable de répondre à notre requête atteint, celui envoie l'adresse IP, mais aussi une durée garantie de validité de cette adresse. Alors *tous* les serveurs de noms et au bout votre machine elle-même vont se souvenir de ce renseignement, pendant un temps fixé, ce qui permet de limiter le trafic sur le réseau⁴⁵.

Évidemment les serveurs DNS contenant des tables associant adresses et noms, la recherche fonctionne en sens inverse : connaissant l'adresse ip, trouver le nom.

45. C'est très utile quand on envisage de changer des adresses : on diminue la durée de validité.

21.4. En résumé. Pour IPv4, il faut au minimum 4 paramètres pour définir la configuration réseau d'une machine :

- L'adresse de la machine.
- Celle de la GATEWAY.
- Celle du DNS.
- Le netmask.

Pour une description plus détaillée consulter par exemple [7].

21.5. IPv6. Les adresses ne sont plus stockées sur 32 bits mais sur 128 bits (16 octets).

Petit calcul :

- le rayon R de la terre est de 6000 kms environ, soit 6000×1000 mètres. la surface de la terre est donc environ (en mètres carrés) de

$$S = 4\pi R^2.$$

- Le nombre d'adresses IP v6 disponibles par mètre carré est donc environ :

$$\frac{2^{128} - 1}{4\pi R^2},$$

ce qui fait environ $7,5 \times 10^{23}$ adresses au mètre carré ⁴⁶ !

La notation des adresses est un peu différente. Cela pourra être :

`2001:0db8:0000:85a3:0000:0000:ac1f:8001`

Pourquoi des lettres (d, b, a) ? Ces adresses sont écrites en base 16 pour laquelle on note les chiffres 0, 1, ..., 9, a, b, c, d, e, f (15 chiffres), ce qui rend l'écriture plus compacte. Des écritures condensées existent, voir [10].

21.5.1. Notes importantes.

- La migration vers v6 est en cours (v6 est disponible chez free.fr et adopté en remplacement de v4 chez orange (peut-être seulement sur certaines parties du réseau ?)).
- IPv4 et IPv6 forment deux réseaux différents qui peuvent communiquer par des passerelles.
- Il est bien évident qu'avec IPv6, le NAT disparaît. Chacune de vos machines, est accessible directement derrière votre box ! (et tous les ports de toutes les machines peuvent à priori être accédés). Toutes vos machines sont directement sur l'internet, ont chacune une adresse IPv6, mais ceci n'empêche pas d'avoir des adresses locales, pas visibles de l'extérieur, c'est à dire correspondant à votre réseau privé.
- Pour le netmask, rien de changé, on adapte juste à la longueur de l'adresse.
- Le DNS est adapté.

46. Comment je calcule ça ? Avec SageMath : <https://www.sagemath.org/> et <http://sagebook.gforge.inria.fr/english.html>. Notons que 10^{23} c'est 100 000 milliards de milliards.

- Si vous êtes sur un réseau en IPv6 (vous pouvez être sur les deux : v4 et v6), vous pourrez vérifier en allant sur <https://www.whatismyip.com/fr/> que des machines différentes ont des adresses IPv6 différentes, même derrière votre box. Mais il se peut aussi que certaines machines de votre réseau (c'est le cas de mon téléphone connecté au wifi de ma box) ne soient pas configurées pour se connecter en IPv6.

21.6. Ports. A chaque adresse IP (v4 ou v6) on associe des ports, qui sont des nombres entiers. Cela permet de séparer le trafic. Chaque protocole est associé à un ou plusieurs ports. Par exemple le port 80 est associé à `http`, et un serveur web écoutera à priori sur ce port.

Dans le cas de ma box, il est possible de *router* un port donné sur vers machine du réseau local, par exemple le port 80 vers la machine 1⁴⁷.

22. Comment explorer tout ça sous Linux ?

22.1. La configuration de ma machine. Dans `/etc`, il y a un certain nombre de fichiers et de répertoires dont le nom commence par `network`. avec la configuration automatique par DHCP, il n'y a plus trop de raisons d'aller les regarder.

Le fichier `/etc/services` est intéressant, même si il n'y a pas vraiment de raisons de le modifier : ce fichier contient, pour « tous » les protocoles réseau le port utilisé (*well known port*) et le protocole de transport utilisé (`tcp` ou `udp`). Voici par exemple les lignes correspondant à `http` et `https`⁴⁸ :

```
http 80/tcp www # WorldWideWeb HTTP
https 443/tcp # http protocol over TLS/SSL
```

Donc `http` utilise le port 80 et `https` le port 443 ; tous deux utilisent `tcp`.

22.1.1. *La commande ip.* La commande :

```
cmd>ip address show
```

ou simplement

```
cmd>ip a49
```

va montrer quelque chose qui ressemble à ceci :

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
   link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
   inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
   inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever

2: enp3s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default qlen 1000
   link/ether d8:50:e6:ba:2e:d8 brd ff:ff:ff:ff:ff:ff
   inet 192.168.0.10/24 brd 192.168.0.255 scope global dynamic noprefixroute enp3s0
       valid_lft 35117sec preferred_lft 35117sec
   inet6 2a01:e34:ec0e:46d0::4ab7:9bd8/128 scope global dynamic noprefixroute
       valid_lft 81139sec preferred_lft 81139sec
```

47. Ce qui restreint les possibilités : un seul serveur web dans le réseau local peut être visible de l'extérieur, par exemple.

48. `cmd>grep http /etc/services`

49. La commande `ip` remplace et généralise la commande `ifconfig`.

```
inet6 fe80::b9fd:1b45:9712:eff9/64 scope link noprefixroute
      valid_lft forever preferred_lft forever
```

Pour une description détaillée de tous ces résultats, voir par exemple [2]. De façon résumée, `lo` et `enp3s0` sont deux interfaces réseau :

- (1) `lo` (loopback) est une interface qui n'est pas *physique*. Elle permet à la machine de se connecter à elle-même, de permettre à des programmes de dialoguer « comme si » c'était à travers un vrai réseau.
- (2) `enp3s0` : ici, une carte ethernet (cela pourrait être une interface wi-fi). Détaillons un peu :
 - `192.168.0.10/24` : c'est l'adresse IPv4, avec le netmask de 24 bits, comme on l'a vu plus haut.
 - `2a01:e34:ec0e:46d0::4ab7:9bd8/128` : c'est une adresse IPv6. On remarque que le netmask est de 128 bits, ce qui veut dire que cette adresse est seule dans son réseau. C'est l'adresse publique, celle qu'on voit quand on va sur le site <https://www.whatismyip.com/fr/>.
 - `fe80::b9fd:1b45:9712:eff9/64` : une adresse IPv6 et son netmask (64 bits de gauche). Comme le netmask est de 64 bits, il reste $128 - 64 = 64$ autres bits libres ($2^{64} \simeq 1.8 \times 10^{19}$ adresses!). Cette adresse n'est pas publique (propriété générale des adresses commençant par `fe80`). On a ainsi un réseau IPv6 local.
 - L'interface physique est donc associée à 3 adresses : une adresse IPv4 et deux adresses IPv6.

On peut se limiter aux renseignements concernant IPv4 (ou IPv6) en utilisant respectivement les commandes :

```
cmd>ip -4 a
```

et

```
cmd>ip -6 a
```

22.1.2. *ping*. Utilisant ICMP (*Internet Control Message Protocol*), le protocole de message de contrôle sur Internet), `ping` envoie une demande d'écho à une machine du réseau. Exemples :

```
cmd>ping berkeley.edu
```

```
ING berkeley.edu (35.163.72.93) 56(84) bytes of data.
```

```
64 bytes from ec2-35-163-72-93.us-west-2.compute.amazonaws.com (35.163.72.93): icmp_\_
```

```
64 bytes from ec2-35-163-72-93.us-west-2.compute.amazonaws.com (35.163.72.93): icmp_\_
```

(Control-C pour interrompre). Mais la machine distante est libre de ne pas répondre ! (ça se configure).

```
cmd>ping localhost utilisera l'interface lo de votre machine (tester).
```

22.1.3. *traceth* (ou *traceroute*). Permet de savoir par où on passe pour atteindre une adresse donnée. Tenir compte du fait que certaines machines sont à l'interface de deux réseaux et ont plusieurs adresses : on obtiendra donc en général plus d'adresses que de machines physiques.


```
cmd>tracertpath aldil.org
```

(à tester pour s'amuser).

22.2. whois. Utile pour savoir qui gère un domaine. Exemple :

```
cmd>whois thierry-dumont.fr
```

22.2.1. *nslookup*. Questionner un DNS :

```
cmd>nslookup aldil.org
```

mais aussi :

```
cmd>nslookup 80.67.185.24
```

22.2.2. *nmap*. *nmap* sert à scanner les ports d'une machine. Exemple :

```
cmd>nmap aldil.org
```

renvoie la liste des ports ouverts sur *aldil.org* :

Not shown: 992 filtered ports

PORT	STATE	SERVICE
22/tcp	open	ssh
25/tcp	open	smtp
80/tcp	open	http
143/tcp	open	imap
443/tcp	open	https
465/tcp	open	smtps
587/tcp	open	submission
993/tcp	open	imaps

— les ports 25, 143, 587 et 993 correspondent au serveur de courrier électronique,

— les ports http et https sont ouverts pour le serveur web,

— sur le port 22, ssh permet une connexion cryptée sur la machine,

bref, que du nécessaire⁵⁰.

22.3. Ouvrir ou fermer des ports : le firewall. Un des intérêts de la notion de port est qu'on peut filtrer intelligemment ceux qui sont ouverts ou fermés. Le firewall est intégré à Linux :

— *Netfilter*[30] est implanté au sein du noyau linux.

— *iptables*[26] est un programme de l'espace utilisateur, pour configurer *netfilter*⁵¹.

On peut utiliser ces outils pour implanter un NAT (voir page 34) : donc vous ne devez pas être très étonné d'apprendre que vos « box » sont des systèmes Linux.

L'utilisation des *iptables*, sans être difficile, est hors du champ de ce cours.

On peut recommander aux utilisateurs qui ne veulent pas investir trop dans les *iptables* de se tourner vers des outils qui en simplifient l'usage (et certainement en restreignent un les possibilités) comme *ufw*[6], simple à utiliser. On recommande toujours de procéder de la manière suivante, *si c'est possible* :

(1) Fermer tous les ports.

(2) Ouvrir ensuite seulement ceux dont on a besoin.

50. La machine de l'Aldil est une machine virtuelle; elle est directement sur l'internet, et pas derrière une quelconque box qui ferait office de filtre

51. Il faut être super-utilisateur pour l'utiliser, bien sûr.

23. Installation de « packages », mises à jour.

Comment installer de nouveaux programmes, de nouveaux outils ? ou les mettre à jour ? Comme Linux est diffusé en *distributions*, il y a deux cas possibles :

- (1) Le programme (au sens large : peut-être est-ce une bibliothèque) est fourni par la distribution, mais pas encore installé.
- (2) Il n'est pas dans la distribution.

24. Installation de packages fournis par la distribution

Dans cette section on regarde le premier cas, avec les outils des distributions « Debian-like » (Debian, Ubuntu, Mint,...).

Il existe des outils graphiques comme **Synaptic**, bien agréables, mais on s'intéresse ici la ligne de commande.

24.1. La commande apt. C'est une version modernisée de la commande **apt-get**.

Rappelons que la distribution Debian a été la première à gérer correctement les dépendances entre packages. De quoi s'agit-il ? La commande **apt show** nous le dit ; exemple :

```
cmd>apt show tesseract-ocr
Depends: libarchive13 (>= 3.2.1), libc6 (>= 2.29), libcairo2 (>=
1.2.4), libfontconfig1 (>= 2.12.6), libgcc-s1 (>= 3.0), libglib2.0-0
(>= 2.12.0), libicu67 (>= 67.1-1~), libjpeg-turbo8 (>= 1.75.3),
libpango-1.0-0 (>= 1.37.2), libpangocairo-1.0-0 (>= 1.22.0),
libpangoft2-1.0-0 (>= 1.14.0), libstdc++6 (>= 5.2), libtesseract4 (=
4.1.1-2build3), tesseract-ocr-eng (>= 4.00~), tesseract-ocr-osd (>=
4.00~)
```

Cela nous dit que le package **tesseract-ocr** dépend (utilise) des bibliothèques (les **lib***), mais aussi d'autres programmes (**tesseract-ocr-eng** et **tesseract-ocr-osd**). Et à chaque fois, la commande nous dit quel numéro de version minimal doit être utilisé.

Donc quand on va installer un package comme **tesseract-ocr**, il faut que le système de gestion de packages installe aussi tous les packages dont il dépend. De même en cas de mises à jour, il faut les propager si nécessaire.

24.1.1. *Principales options de la commande apt.*

- **apt search**. Plusieurs possibilités :
 - **apt search tesseract-ocr** par exemple.
 - **apt search "tesseract*"** (tous les packages dont le nom commence par **tesseract** ; on peut mettre n'importe quelle *expression régulière*).
- **apt show** vu ci-dessus.
- **apt install** : installer un package. Exemple :

```
cmd>apt install tesseract-ocr
```

installe le package et ses dépendances (plus exactement : les dépendances d'abord, puis le package).

- **apt remove** et **apt purge** : supprime un package. Avec l'option **purge**, la suppression est radicale : on efface tous les fichiers de configuration. Attention, on ne supprime pas les dépendances. Pour supprimer les dépendances, il faut utiliser la commande **apt autoremove**.

24.1.2. *Les mises à jour.* Il faut utiliser successivement 2 commandes :

- (1) **apt update** : met à jour la liste des packages. On va pouvoir ensuite comparer les numéros de version disponibles avec ce qui est installé. Alors, la commande :
- (2) **apt upgrade**. Met à jour ce qui doit l'être.

Pour la mise à jour ou l'installation de packages : les packages sont d'abord téléchargés (fichiers avec l'extension **.deb**), puis installés. La suppression des **.deb** n'est *pas automatique* : il convient de lancer la commande **apt clean** après une installation ou une mise à jour.

Notons aussi que dans une mise à jour il y a non seulement installation des nouvelles versions des packages, mais redémarrage des services qui y sont associés, s'il en existe. Par exemple, le démon **systemd** sera relancé après une mise à jour du programme (du fichier) **systemd**.

24.2. Faut-il rebooter la machine ? Oui, dans certains cas, en particulier quand le noyau a été mis à jour (package **linuximage**). Les interfaces graphiques l'indiquent, pas celles en ligne de commande. Après le redémarrage (car c'est seulement après le redémarrage qu'on va utiliser le nouveau noyau), il convient de lancer la commande

```
cmd>apt autoremove
```

pour supprimer l'ancienne version du noyau.

25. Installer depuis la « source »

On peut avoir besoin d'installer un programme ou une bibliothèque qui n'est pas disponible comme package pré-compilé dans une distribution (ou aussi si on veut une version plus récente que celle disponible).

Il y a deux cas un peu différents :

- (1) Le cas d'un programme écrit dans un langage compilé (C, C++,...).
- (2) Le cas d'un programme écrit dans un langage interprété, le cas emblématique étant Python.

On regarde ici le premier cas.

Les codes source sont en général stockés dans un répertoire ; ce répertoire contient en principe :

- Des fragments de code (fichiers d'extensions : **.h**, **.c**, **.cc** etc.).
- Des procédures pour compiler (traduire) le code en langage machine et pour l'installer dans l'arborescence de la machine.
- Éventuellement de la documentation.
- Des modes d'emploi pour l'installation (fichiers **README**, **INSTALL** etc.)

Dans une première étape, on récupère le répertoire, qui contient le code source, ensuite on *compile*, puis on installe le programme. Commençons par le processus de compilation et d'installation.

25.1. Le processus de compilation. C'est un processus complexe, mais on va l'automatiser.

25.1.1. *La compilation proprement dite.* Cela consiste à traduire chaque fragment de code (admettons que c'est écrit en langage C, mais peu importe⁵²) en langage machine : à partir du fichier, disons `toto.c`, on crée le fichier `toto.o`. C'est un programme, le *compilateur*, qui fait ça. La commande à la main serait :

```
cmd>gcc toto.c -o toto.o53
```

La commande `gcc` est en général accompagnée d'options pour mieux optimiser le code engendré, l'adapter précisément au type de cpu de la machine⁵⁴, ou à une syntaxe particulière. Un exemple typique de ligne de compilation pourra être :

```
cmd>gcc -O 3 -m cpu=native toto.c -o toto.o55
```

25.1.2. *L'édition des liens.* Une fois tous les fragments de programme traduits (compilés), il faut les relier entre eux, leur associer des liens vers des bibliothèques existantes, etc. C'est un travail assez complexe, fait par la commande `ld` (que l'utilisateur, même s'il développe des programme, n'a pas en général à connaître). Le résultat est un fichier, qui contient du langage machine, prêt à être exécuté⁵⁶. Rappelons que l'exécution de ce programme consistera entre autres, à recopier ce fichier en mémoire.

25.1.3. *Installer le programme.* Si tout s'est bien passé, il faut ensuite installer le programme créé (il peut y en avoir plusieurs!) dans un endroit accessible (dans le PATH des utilisateurs), c'est à dire copier un ou plusieurs fichiers dans un répertoire (dont le nom se termine par `bin` en général), et leur donner les bons droits (`x` pour tout le monde).

On peut bien sûr, ensuite, se débarrasser du répertoire source.

25.1.4. *Un processus trop complexe, et les remèdes.* Il faut imaginer que le code source peut être éclaté en des centaines, des milliers ou plus de fichiers. Un certain nombre de techniques vont simplifier et automatiser son installation.

(1) La commande `make` (voir [28]).

Elle lit un fichier de nom `Makefile` (par exemple) et fait... ce qu'il faut, ou plutôt ce qu'on lui dit de faire.

Exemple de fichier `Makefile` :

```
toto.o toto.c
gcc toto.c -o toto.o
tutu.o tutu.c machin.h
gcc tutu.c -o tutu.o
prog: toto.o tutu.o
ld ...
```

Ce qui se lit ainsi (commencer par le bas) :

52. Rappelons que C est le langage de base des systèmes Unix : il a été créé pour développer Unix.

53. `gcc` est le compilateur standard sous Linux ; sous les systèmes à la *Debian*, il peut être installé par la commande `apt install build-essential`. Mais il existe d'autres compilateurs du langage C dans les distributions Linux, comme le compilateur `clang`.

54. Ce qui va probablement améliorer les performances, mais rendre très probablement le code binaire non portable d'une machine à l'autre.

55. Ce qui veut dire : optimiser au niveau 3 (fortement), et engendrer du code pour le cpu local.

56. à la différence des fichiers comme `toto.o` à qui il manque beaucoup de choses, comme les appels de bibliothèque, pour simplifier.

- Pour créer le programme `prog`, il faut auparavant créer `toto.o` et `tutu.o` et effectuer la commande `ld` qu'on ne détaille pas ici.
- Pour obtenir le fichier binaire `toto.o`, il faut compiler `toto.c` (deux premières lignes, la deuxième ligne indique ce qu'il faut faire pour cela). Idem pour obtenir `tutu.o`, il faut compiler `tutu.c`; ici on a une seconde dépendance, à partir de `machin.h`.

Avantage de cette technique : si le développeur change disons `toto.c`, on n'a pas à recompiler `tutu.c`, mais seulement `toto.c` (mais il faut recréer `prog` à partir des différents fichiers `*.o` (deux dernières lignes du fichier `Makefile`)). C'est très important : certains projets comportent des milliers de fichiers source !

- (2) Bien. Mais la création du fichier `Makefile` s'avère rapidement cauchemardesque. Pour automatiser son écriture, on a créé un ensemble de programmes, les `autotools` (voir [17]). C'est fort complexe, mais il n'est pas nécessaire de les maîtriser si on n'est pas développeur⁵⁷.

Les `autotools` engendrent un fichier `Makefile`, à partir de fichiers de configuration, réputés être simples (ça se discute), et permettent aussi de tenir compte de particularités de la machine et du système (suis-je sous Linux ? sous Windows ? etc.).

25.1.5. *Concrètement, pour l'utilisateur.* Pour compiler / installer un programme, l'utilisateur doit seulement :

- (a) Lancer un ou plusieurs des `autotools`; en général, il suffit de lancer :
- ```
cmd>./configure
```
- dans le répertoire source (C'est en général documenté (fichier `INSTALL` ou/et `README`) dans le répertoire source).
- cette commande va engendrer le fichier `Makefile` *si c'est possible*. Mais peut-être que des bibliothèques, des programmes vont manquer : le script `./configure` va vous le dire; il faudra les installer pour aller plus loin (avec un peu de chance, ce sont des packages de la distribution qui doivent être installés), puis relancer `./configure` (recommencer éventuellement jusqu'à ce que toutes les dépendances nécessaires soient satisfaites).
- (b) Taper `make`<sup>58</sup>
- et attendre que la compilation soit terminée.
- (c) Si tout s'est bien passé, il faut lancer la commande `cmd>make install` ou plutôt `cmd>sudo make install`.

**Remarque :** sous Ubuntu, il faut au moins installer les paquets `build-essential` et `autotools-dev` pour pouvoir compiler un programme.

## 25.2. Récupérer le répertoire source. Deux possibilités :

- (1) Fichier `tar`, en général compressé.
- (2) « Repository » `git`.

<sup>57</sup>. Des alternatives existent, plus simple comme `cmake` (voir [20]). `cmake` est développé par la société kitware, spécialiste de visualisation scientifique, présente à Villeurbanne.

<sup>58</sup>. On peut essayer `make -j n` avec pour `n` le nombre de cœurs, mais ça ne fonctionne pas toujours !

25.2.1. *La commande `tar`*. Comment mettre un répertoire entier dans un seul fichier ? Il faut utiliser la commande `tar`.

(1) Pour mettre un répertoire dans un fichier, on fait :

```
cmd>tar cf xxx.tar répertoire
```

Exemple : `cmd>tar cf tesseract.tar tesseract`

Tout le répertoire `tesseract` se retrouve stocké dans le fichier `tesseract.tar`.

(2) L'opération inverse est :

```
cmd>tar xf xxx.tar
```

Exemple : `cmd>tar xf tesseract.tar`.

En général, on compresse le fichier `tar` (on le crée compressé) :

```
cmd>tar zcf tesseract.tar.gz tesseract
```

et l'opération inverse est :

```
cmd>tar zxf tesseract.tar.gz
```

qui extrait le répertoire contenu dans le fichier `tar`.

Donc si on récupère un fichier `.tar.gz`, on refabrique un répertoire.

Notons que `tar` est pratique pour échanger des répertoires entiers, par mail ou par tout autre moyen.

25.2.2. *L'alternative moderne : `git`*. `git` (voir [24]) est outil de développeur. Il permet (entre autres) :

- un développement collaboratif,
- une gestion fine des historiques, des versions successives, des « forks ».

Écrit par Linus Thorvald pour la maintenance du noyau Linux, il est devenu le standard de fait des développeurs « libres » et *aussi un moyen de distribuer les logiciels libres*. C'est un logiciel relativement complexe pour l'utilisation comme développeur, mais simple à utiliser pour ce qui nous intéresse : installer des logiciels depuis leurs sources.

Quelques entreprises, en particulier `github`<sup>59</sup> ont développé tout un système autour de `git`. `gitlab` propose à peu près la même chose, mais leur logiciel est libre.

Le logiciel libre dépend *très fortement* de `github`.

Parmi les ajouts de `github` (et `gitlab`) à `git` :

- L'intégration continue : à chaque *push* d'un contributeur (envoi d'un nouveau fichier : correction ou nouveauté), on peut déclencher des actions : recompilation, tests, etc.
- Un Wiki par « projet ».
- Des outils pour que des tiers puissent proposer des modifications.
- Des appréciations (*star*).
- etc.

---

59. Racheté par Microsoft...

25.2.3. *Utiliser git pour installer du logiciel.* Il faut d'abord installer `git` sur la machine :

```
cmd>apt install git
```

sur les systèmes de type debian.

Ensuite, connaissant l'url du *repository*, la commande (exemple) :

```
cmd>git clone https://github.com/tesseract-ocr/tesseract.git60
```

clone le répertoire (ici : `tesseract`).

Y a t-il eu des mises à jour depuis que j'ai cloné le répertoire ? (la question peut se poser un certain temps si le projet est actif). La commande :

```
cmd>git pull
```

téléchargera les mises à jour. Il faut bien sûr recompiler le code, ce qui en général se fait simplement par :

```
cmd>make
```

et :

```
cmd>make install61
```

La documentation est, en principe, dans le répertoire.

## 26. Problématique

On va supposer que deux personnes veulent dialoguer ; il est habituel de les appeler Alice et Bob.

Pour pouvoir communiquer de manière sécurisée il faut :

- (1) Crypter les messages.
- (2) S'assurer que, quand Alice écrit à Bob (et vice-versa), c'est bien Bob son interlocuteur et non pas un usurpateur.

### 26.1. Comment crypter les messages ? Il faut :

- (1) Définir un système de cryptographie qu'on puisse considérer comme indéchiffrable.
- (2) Utiliser ce système de façon sûre.
- (3) Il faut aussi que crypter et décrypter ne soit pas trop *coûteux* car on veut pouvoir échanger de grands messages (des fichiers de grande taille, par exemple).

Les systèmes de cryptographie sont tous basés sur deux composants :

- (1) Un algorithme.
- (2) Une ou plusieurs « clés » qui paramètrent cet algorithme.

Les systèmes de cryptographie ont longtemps eu un problème au niveau de la clé de chiffage/déchiffage : les systèmes classiques sont basés sur une seule clé qui sert à la fois au chiffage et au déchiffage ; on parle alors de cryptographie symétrique. Si Alice et Bob veulent discuter, cela nécessite qu'ils partagent la même clé. Le partage de la clé est resté une faiblesse des meilleurs systèmes, comme la machine Enigma allemande pendant la deuxième guerre mondiale : la capture d'un sous-marin avec une valise de

60. `github` et `gitlab` proposent de récupérer l'url en un clic, sur la page web associée à chaque *repository*.

61. avec `sudo` ou l'équivalent devant.

clés (les sous-marins portaient pour longtemps et, par prudence, il fallait changer de clé tous les jours) a grandement fait avancer la décryptage d'Enigma par les spécialistes de Bletchley-Park. En temps de paix, jusqu'aux années 80, des personnels spécialisés parcouraient le monde pour transporter les clés, prenant moult précautions. On verra qu'il est possible de partager une clé de manière sûre.

**26.2. La cryptographie à clés asymétriques.** Ça a été le bond en avant de la cryptographie.

- (1) L'algorithme utilisé pour crypter les messages par Bob et Alice utilise non pas une mais *deux* clés différentes : une pour chiffrer un message et l'autre pour le déchiffrer.
- (2) Une des deux clés ne peut pas être déduite de l'autre.

Ainsi Bob possède deux clés :

- Une est la clé *privée*, qu'il garde précieusement cachée. C'est elle qui va servir à déchiffrer les messages.
- L'autre est *publique*, qu'il peut diffuser autant qu'il veut sans précaution particulière.

La clé *privée* est celle qui ne peut pas se déduire de la clé *publique*.

Supposons qu'Alice veuille écrire à Bob ; elle le lui fait savoir par un message envoyé « en clair ». Alors :

- Bob envoie sa clé *publique* à Alice ;
- Alice se sert de cette clé pour chiffrer le message envoyé à Bob.
- Bob se sert de sa clé *privée* pour déchiffrer le message d'Alice.

Pour que ça marche, c'est à dire pour que ça garantisse la confidentialité du message envoyé par Alice, il faut absolument qu'on ne puisse pas *déduire* la clé *privée* de Bob de sa clé *publique* (qu'il a donnée à Alice). Mais alors, la clé publique peut être diffusée autant qu'on veut, sans risque : elle ne sert qu'à crypter, et il faut la clé privée pour décrypter.

Bien sûr, dans l'autre sens, Alice renvoie sa clé *publique* à Bob, qui s'en sert pour lui écrire, etc.

**26.3. Le problème de la cryptographie asymétrique.** On verra plus loin comment on réalise des systèmes cryptographiques asymétriques ; mais avant, il faut parler de leur principal défaut : ils sont trop coûteux pour traiter de grandes quantités d'information.

On contourne ce problème en n'utilisant le système à clés asymétriques que pour échanger secrètement LA clé d'un système de cryptographie symétrique : on établit une connexion avec un système de chiffrement asymétrique : ce système est utilisé seulement pour l'échange de LA clé d'un système de cryptage *symétrique*. Remarquons que le problème du partage de la clé, qui était le point faible de tous les systèmes de cryptographie symétrique est résolu<sup>62</sup>.

On limite en général la « durée de vie » de cette clé et, si nécessaire, on fabrique au bout d'un temps fixé une nouvelle clé pour de chiffrement *symétrique* et on l'échange avec le système de cryptage asymétrique.

62. Mais en utilisant une technologie plus sophistiquée : le cryptage asymétrique !



Il faut « seulement » que le système de cryptage *symétrique* soit assez robuste pour ne pas être décrypté par quelqu'un qui ne possède pas la clé.

## 27. Un exemple : ssh (Secure Shell)

**ssh** est à la fois un ensemble de programmes et un protocole (une norme) pour la communication cryptée. Dans sa version la plus populaire il permet de d'ouvrir un shell sur une machine lointaine :

```
cmd>ssh monLogin@machinelointaine.domaine
```

Cette commande :

- (1) Ouvre une connexion cryptée asymétrique : le mot de passe est donc crypté.
- (2) Ensuite, on passe à une connexion cryptée, symétrique.

D'autres commandes en dérivent, par exemple :

- **scp** : copie cryptée entre machines.
- **sftp** : comme **ftp**, mais crypté.

On peut se servir de **ssh** d'un grand nombre de manières : par exemple, avec **rsync** pour synchroniser des répertoires entre deux machines distantes (pratique pour faire des backups très sûrs). L'implantation de **ssh** qui tourne sous Linux est **openssh**.

Sur le serveur (la machine à laquelle on se connecte), doit tourner un démon **sshd**. Lors de l'installation de ce logiciel, une paire de clés (privée, publique) propre à la machine est engendrée<sup>63</sup>.

### 27.1. Quelques détails à propos de openssh.

27.1.1. *Côté serveur*. C'est donc le démon **sshd** qui tourne. Sa configuration est dans `/etc/ssh` :

- Le fichier **sshd\_config** contient la configuration. Les valeurs par défaut sont en général suffisantes, mais on peut par exemple, changer le port (22 par défaut) et obliger l'utilisation d'une connexion par clé (voir plus loin) ou interdire le login **root**.
- Plusieurs fichiers dont le nom commence par **ssh\_host** contiennent les clés privées et publiques (suffixe **.pub**) du serveur. Pourquoi plusieurs ? Parce que **ssh** peut utiliser plusieurs protocoles de cryptographie. C'est quelque chose qui se négocie lors de la connexion du client.

**Important** : en cas de réinstallation du système, il est recommandé de sauvegarder ces clés et de les réinstaller sur le nouveau système. Sinon, on souffre quelques désagréments.

27.1.2. *Côté client*. La configuration est dans le répertoire caché **.ssh/**, dans le home directory. Elle est créée lors de la première utilisation. On y trouve un fichier **config**. A priori, il n'y a pas grand chose à y mettre, il doit être vide par défaut.

---

63. Ce qui implique un tirage de nombres au hasard.

## 28. Cryptographie : méthodes

Les problèmes de cryptographie reposent tous sur une branche des mathématiques désignée sous le nom de *théorie des nombres*<sup>64</sup>. Parmi les domaines d'études de la théorie des nombres figure tout ce qui touche aux nombres premiers<sup>65</sup>.

**28.1. Cryptographie symétrique.** Il faut s'assurer que quelqu'un qui ne connaît pas la clé ne pourra pas la découvrir. Un des standards actuels est AES (voir [15] (assez technique)).

**28.2. Cryptographie asymétrique.** Tous les algorithmes de cryptographie asymétrique sont basés sur le même principe :

- (1) Fabriquer la clé *privée* est un problème facile, c'est à dire un problème qu'on peut résoudre en un temps calcul très limité.
- (2) La clé *publique* est fabriquée à partir de la clé privée, là aussi en résolvant un problème peu coûteux.
- (3) **Mais le calcul inverse**, c'est à dire celui de retrouver la clé privée à partir de la clé publique, s'il est théoriquement possible, demande un temps calcul totalement inaccessible.

Un autre propriété souhaitable est l'adaptabilité : même si les ordinateurs et les algorithmes disponibles pour *attaquer* la méthode progressent, on doit pouvoir en rester maître en n'augmentant que peu la complexité des calculs nécessaires pour pouvoir continuer à crypter et à décrypter les messages sans risques.

**28.3. Complexité d'un algorithme.** La complexité d'un algorithme est le nombre d'opérations qu'il met en œuvre, en fonction de la taille du problème à résoudre. Voici un exemple assez simple :

28.3.1. *Complexité de la multiplication.* On a tous appris à faire des multiplications : Quelles sont les opérations ? Il y a (exemple à gauche) :

|       |   |   |   |   |
|-------|---|---|---|---|
|       | 2 | 3 | 1 |   |
| ×     | 1 | 2 | 4 |   |
| <hr/> |   |   |   |   |
|       | 9 | 2 | 4 |   |
|       | 4 | 6 | 2 |   |
| 2     | 3 | 1 |   |   |
| <hr/> |   |   |   |   |
| 2     | 8 | 6 | 4 | 4 |

- des multiplications entre chiffres ( $4 \times 1$ ,  $4 \times 3$  etc.),
- des additions.

Ici on fait  $3 \times 3$  multiplications entre chiffres. Les additions sont au maximum de 8, car il peut y avoir des retenues.

Mais ce qu'on veut, c'est savoir comment cela se comporte quand on multiplie des nombres de  $n$  chiffres ( $n = 4, 5, 6, \dots, 100000, \dots$ ) entre eux. On veut une estimation valable pour tout  $n$ .

Il est assez facile de compter les multiplications entre chiffres : il y en a exactement  $n^2$  (ça se vérifie facilement sur l'exemple ci-dessus pour  $n = 3$ ). Pour les additions, c'est un

64. Il y a quelque chose de piquant là-dedans : avant l'apparition des méthodes de cryptographie modernes, les théoriciens des nombres disaient à-peu-près « Ça ne sert à rien, et donc les militaires ne vont pas se servir de mes recherches ».

65. Rappelons : un nombre entier est premier s'il n'est pas divisible par un nombre plus petit que lui. Par exemple : 2, 3, 5, 11, 13, 17 sont premiers, mais pas 4, 8, 15 etc. La suite des nombres premiers est infinie et donc il en existe d'aussi grands qu'on veut.

peu plus délicat, mais il y en a au maximum  $n^2 - 1$  (ça dépend du nombre de retenues), soit au total (additions et multiplications) de l'ordre de  $2 \times n^2$  opérations<sup>66</sup>.

Implication pratique : ce coût implique simplement que, *quand on double le nombre de chiffres, le nombre d'opérations est multiplié par 4*. En général on dira que la complexité d'un algorithme est, par exemple, en  $n^2$ , négligeant le facteur multiplicatif (2 dans ce qu'on a vu), car ce qui compte, c'est bien comment la difficulté croît quand  $n$  augmente.

***Le coût de la multiplication de deux nombres de  $n$  chiffres est de l'ordre de  $n^2$ .***

28.3.2. *Comparer des algorithmes.* Il doit être clair que, si pour résoudre le même problème on a deux algorithmes de complexités différentes (par exemple  $n^3$  et  $n^2$ ), le plus rapide ( $n^2$ ) est le plus intéressant.

28.3.3. *Les problèmes trop coûteux.* Évidemment, si les seuls algorithmes dont on dispose pour résoudre un problème ont un coût qui croît très vite quand la taille du problème augmente, le problème devient difficile, voire même impossible à résoudre.

Parmi ces problèmes les *pires* sont ceux qui ont un coût dit *non polynomial*. Pour ces algorithmes, et pour une valeur  $p$  *quelconque*, si  $n$  est assez grand, alors le coût devient supérieur à  $n^p$ . De façon plus explicite, si  $n$  est assez grand, le coût deviendra supérieur disons à  $n^{50000}$  ou pour une autre valeur de  $n$ , il deviendra aussi supérieur à  $n^{100000}$  etc.

Parmi les possibilités, une est que le coût soit exponentiel (mais ce n'est pas la seule possibilité). L'exponentielle de  $n$  est<sup>67</sup> :

$$\exp(n) = 1 + n + \frac{n^2}{2} + \frac{n^3}{2 \times 3} + \frac{n^4}{2 \times 3 \times 4} + \cdots + \frac{n^p}{p!} + \frac{n^{p+1}}{(p+1)!} + \cdots,$$

avec  $p! = 1 \times 2 \times 3 \times \cdots \times (p-1) \times p$ .

Si vous comparez  $\exp(n)$  à  $n^p$  vous voyez que  $\frac{n^{p+1}}{(p+1)!}$  divisé par  $n^p$  qui est égal à  $\frac{n}{(p+1)!}$ , devient aussi grand qu'on veut (c'est proportionnel à  $n$ ). On dit alors que l'exponentielle croît plus vite que tout polynôme.

*Les problèmes qui ont un coût non polynomial ne peuvent pas être résolus, dès que la taille  $n$  du problème devient grande.*

Ce qui est une catastrophe dans la vie courante est à la base des méthodes modernes de cryptographie.

**28.4. Une idée de cryptographie asymétrique.** C'est l'idée qui est derrière le système RSA (Ronald Rivest, Adi Shamir et Leonard Adleman, 1983), mais la réalisation est plus complexe que ce qui est expliqué ci-dessous (pour une explication détaillée —technique—, voir [19])<sup>68</sup>.

- (1) On choisit deux *grands nombres premiers*  $a$  et  $b$  (par exemple, chacun ayant plus de 120 chiffres). Curieusement, c'est facile : on a un algorithme rapide pour cela. Ces deux nombres définissent la **clé privée**.

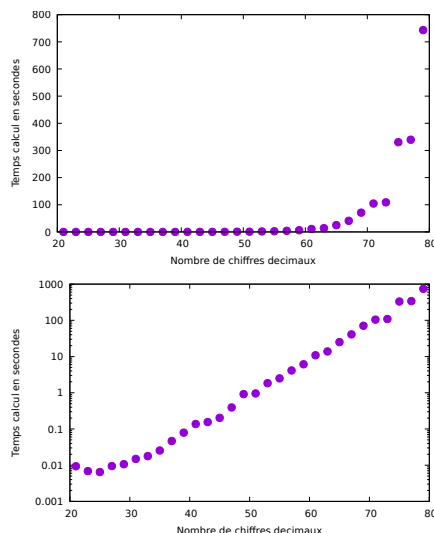
66. Remarquez que le produit de deux nombres d'un million de chiffres demande environ 2000 milliards d'opérations ( $2^{12}$ ) ; avec une machine qui tourne à 3 Ghz, cela va demander environ  $2 \times 10^{12} / (3 \times 10^9) \simeq 666$  secondes. *Mais on peut faire une partie importante du calcul en parallèle (comment ?) et gagner un facteur proche de 4 avec 4 cœurs, par exemple.*

67. Pour en finir avec l'exponentielle à toutes les sauces, voir [1].

68. RSA est considéré comme dépassé, mais reste un très bon exemple.

- (2) On les multiplie entre eux. En utilisant l'algorithme de multiplication classique, pour des nombres de 120 chiffres, cela coûte environ  $2 \times 120 \times 120 = 28800$  opérations, autant dire rien ou presque<sup>69</sup>. **C'est ce nombre qui va définir la clé publique.**

Car la factorisation de  $p = a \times b$ , c'est à dire *retrouver*  $a$  et  $b$  connaissant leur produit  $p$  est un problème difficile, à coût non polynomial.



Les graphiques ci-contre montrent le temps calcul (sur Intel I5, 3,4 Ghz) d'une factorisation d'un nombre de  $n$  chiffres ( $n \leq 80$ ), produit de deux nombres premiers de  $n/2$  chiffres. Le second graphique utilise une échelle logarithmique en ordonnées pour présenter les mêmes résultats. Le calcul a été effectué avec le logiciel **SageMath** [4] qui utilise la bibliothèque **Pari** (Bordeaux, [3]).

#### 28.4.1. Quelques remarques.

- Il n'existe aucune preuve du fait que la factorisation de deux nombres est un problème à coût non polynomial. Mais aucun algorithme n'a jamais été trouvé qui l'infirmes et ce n'est pas faute d'avoir cherché.
- Il existe une compétition de factorisation (voir [31]), une liste de nombres produits de deux nombres premiers, de taille croissante à factoriser. Le record actuel est le nombre RSA-250 (250 chiffres décimaux), factorisé le 28 février 2020 par une équipe de l'INRIA à Nancy. Si vous arrivez à factoriser RSA-1024 (1024 chiffres décimaux), vous gagnerez 100 000 \$.
- Les ordinateurs quantiques, s'ils existent un jour, pourront factoriser des nombres de  $n$  chiffres avec un coût de l'ordre de  $n^3$ , ce qui invalidera les méthodes de cryptographie basées sur ce genre de principe<sup>70</sup>.
- Le principe est une chose, sa réalisation pratique en est une autre. On n'a aucune preuve que **openssh** n'a pas de trou de sécurité ni de bug. Un a été découvert il y a quelque temps : le générateur aléatoire qui choisit les nombres premiers  $a$  et  $b$  était mal utilisé et il les sélectionnait dans un petit ensemble de nombres ce qui, théoriquement, aurait pu permettre de casser le chiffre RSA (avec une très, très grosse puissance de calcul).

69. Et si on veut multiplier entre eux des nombres vraiment très grands, il existe des algorithmes beaucoup plus rapides : le plus simple est l'algorithme de Karatsuba (voir [16]) ; les plus performants utilisent la plus grande invention algorithmique du XX<sup>e</sup> siècle, la Transformée de Fourier Rapide.

70. Pour l'instant les « ordinateurs quantiques » ont réussi à factoriser... 15 (ils ont bien trouvé que  $15 = 5 \times 3$ ). Il reste donc un peu de temps avant de changer de méthode

## 29. Est-ce que je communique avec le bon interlocuteur ?

On sait donc chiffrer les communications. Mais il reste à résoudre le deuxième problème, à savoir s'assurer que son interlocuteur est bien celui voulu.

**29.1. Le cas de ssh (et de ses dérivés).** `ssh` laisse à l'utilisateur le soin de vérifier que l'interlocuteur est bien le bon, mais en l'aidant pas mal.

- (1) A la première connexion sur une machine distante<sup>71</sup>, on reçoit ce genre de message :

```
The authenticity of host 'ssh-math.univ-lyon1.fr (134.214.156.14)'
can't be established.
```

```
Are you sure you want to continue connecting (yes/no)?
```

Si on répond `yes`, la connexion se poursuit avec une demande de mot de passe, etc (rappelons que tout est crypté).

Un fichier `known_hosts` est créé (ou rallongé) dans le répertoire `.ssh/` de votre home directory pour stocker non pas la clé publique du serveur, mais une *empreinte numérique (hash)* de cette clé et du nom du serveur (on va bientôt voir ce que ça veut dire).

- (2) Aux connexions suivantes, `ssh`, en consultant le fichier `known_hosts`, vérifie que rien n'a changé du côté du serveur. Il vous prévient d'un grave danger si quelque chose a changé : peut-être que vous dialoguez avec une machine qui n'est pas celle que vous croyez<sup>72</sup>. Il vous indique quoi faire si vous voulez persister dans votre volonté de vous connecter.

On voit donc que c'est l'utilisateur qui doit s'assurer de l'identité exacte du serveur sur lequel il se connecte. On peut quand même ajouter que l'utilisateur doit avoir un compte et donc un mot de passe sur le serveur pour pouvoir achever la connexion. Évidemment, cette technique n'est pas adaptable à un service web.

29.1.1. *Connexion ssh par clés (sans mot de passe).* C'est très utile, surtout quand on veut faire des connexions automatisées (sauvegardes par `rsync`, par exemple).

On est sur la machine A, et on veut se connecter par `ssh` sur la machine B.

Sur A donc, on tape :

```
cmd>ssh-keygen -t rsa (en répondant oui, ou rien). Cela va créer deux fichiers
dans le répertoire .ssh/ de votre home directory sur A :
```

- `id_rsa` : une clé privée, à ne pas divulguer.
- `id_rsa.pub` : la clé publique correspondante.

Ensuite, on tape :

```
cmd>ssh-copy-id utilisateur@B
```

où "utilisateur" est votre identifiant sur la machine B.

Cela va recopier la clé publique `id_rsa.pub` dans le fichier `.ssh/authorized_keys` de votre home directory, en tant qu'utilisateur "utilisateur" sur la machine B.

Ensuite, vous n'aurez plus besoin de donner votre mot de passe !

71. Ma commande est : `cmd>ssh dumont@math.univ-lyon1.fr`

72. *a man in the middle ?*

**29.2. Empreinte numérique (*hash*).** On aimerait, pour n’importe quel fichier, n’importe quelle suite de caractères pouvoir créer un identifiant *unique*, de taille (nombre de signes) constante et petite.

C’est évidemment impossible : l’identifiant créé étant plus petit que le fichier identifié, il existe forcément des fichiers différents qui ont le même identifiant. Toutefois on peut fabriquer une méthode qui tout en fournissant un résultat de petite taille (un petit nombre de caractères), dépend de façon très *chaotique* de ce qui lui est donné, de sorte qu’une infime modification des données entraîne une profonde modification du résultat.

29.2.1. *Un exemple.* On crée un fichier `c.txt` qui contient juste la chaîne de caractères :

Ceci est le cours Linux de l’Aldil

J’utilise le programme `sha256` (voir [32]) pour en fabriquer l’empreinte :

```
>sha256sum c.txt
6d366b28d175b579ebe8f65b40c4fbe5e3514c45b476bf1c70f72092f0b9e9e6 c.txt
```

La longue chaîne de caractères produite est l’empreinte.

J’ajoute un “.” à la fin de la ligne :

Ceci est le cours Linux de l’Aldil.

```
>sha256sum c.txt
b8400e182df47650b59e332a8077f0852edbf1005984738dc7397d320edb8901 c.txt
```

petite modification, grands effets.

29.2.2. *Quelques méthodes.*

- `md5sum`, considéré aujourd’hui comme trop faible pour des applications critiques.
- La famille `sha2` (`sha224sum`, `sha256sum`, `sha384sum`, `sha512sum`) : celle qu’il est recommandée d’utiliser. La différence entre les versions tient à la longueur du condensat produit et à la taille des blocs sur lesquels on opère.
- La famille `sha3` : encore plus sûre que les `sha2`, mais peu utilisée à l’heure actuelle (`sha384sum` est disponible sur certaines distributions).

29.2.3. *Comment ça marche ?* De manière très schématique, on regarde le matériel à hacher (fichier, texte,...) comme une suite de bits, qu’on découpe en morceaux de taille fixe (quitte à compléter par du remplissage). Supposons que le hachage ait une taille de 256 bits (SHA-256). Alors, de manière simplifiée (voir [32] pour une description plus détaillée, mais pas forcément très claire) :

- (1) On initialise ce qui sera le résultat final  $\mathcal{H}$ , un bloc de 256 bits, avec une constante prédéfinies, qui fait partie de la méthode.
- (2) Ensuite, pour chaque morceau  $\mathcal{B}$  du fichier à hacher, on applique un certain nombre de transformations qui font interagir  $\mathcal{H}$  et  $\mathcal{B}$  pour obtenir une nouvelle version de  $\mathcal{H}$ . Les transformations sont :
  - des “et” logiques (1 = vrai, 0 = faux) <sup>73</sup>.
  - des “ou” logiques <sup>74</sup>.

73. vrai et faux = faux, vrai et vrai = vrai, faux et faux = faux.

74. vrai ou vrai = vrai, vrai ou faux = vrai, faux ou faux = faux.

- des rotations appliquées aux blocs de bits <sup>75</sup>.
- etc. A la fin, on renvoie  $\mathcal{H}$ .

C'est une idée assez voisine de *mélange* (mélanger de la peinture blanche et de la peinture noire) : on veut créer un désordre irréversible.

**29.3. Garantir l'intégrité d'un message.** Bob envoie un message à Alice. Comment faire pour que Alice puisse s'assurer que le message n'a pas été modifié ?

On suppose que Bob possède une paire de clés (clé publique, clé privée) d'un algorithme de cryptographie asymétrique et que Alice en possède la clé publique. Alors :

- (1) Bob calcule l'empreinte (`sha256sum` ou autre) de son message.
- (2) Il crypte cette empreinte avec sa clé privée.
- (3) Il envoie le couple (message, empreinte cryptée) à Alice.

Alice reçoit le message et l'empreinte cryptée du message. Elle va :

- (1) Recalculer l'empreinte du message qu'elle a reçu. Appelons "E1" le résultat.
- (2) Décrypter l'empreinte cryptée envoyée par Bob avec la clé publique de Bob, qu'elle possède. Elle obtient "E2".
- (3) Elle compare E1 et E2. S'ils sont égaux, c'est bien que :
  - (a) le message n'a pas été modifié.
  - (b) l'empreinte a bien été cryptée avec la clé privée de Bob.

Mais ça ne suffit pas à tout assurer : le point faible, c'est évidemment l'échange de clé : comment s'assurer que la clé publique que possède Alice est bien celle de Bob ? Il y a plusieurs possibilités :

- (1) Une rencontre entre Bob et Alice (à supposer qu'ils se connaissent) pendant laquelle Bob donne sa clé publique à Alice.
- (2) Que la clé soit stockée chez un tiers de confiance.

**29.4. Certificats et tiers de confiance.** Les tiers de confiance s'appellent *Autorité de Certification*. Ils délivrent des *certificats*.

29.4.1. *Certificats*. Ce sont des fichiers à la structure normalisée (norme X-509). Un certificat contient principalement :

- Un numéro de version.
- Un numéro de série.
- L'algorithme de signature du certificat.
- DN (Distinguished Name) <sup>76</sup>.
- Validité (dates limites : pas avant, pas après).
- DN de l'objet du certificat.
- Informations sur la clé publique :

<sup>75</sup>. Par exemple, pour une rotation à droite de  $n$  bits, les  $n$  bits qu'on pousse en dehors du bloc à droite rentrent à gauche.

<sup>76</sup>. Des informations du genre « état civil » : Nom, adresse, etc.

- L'algorithme de la clé publique.
- La clé publique proprement dite.
- Et des champs optionnels.

**et** la signature des informations ci-dessus par l'autorité de certification : une empreinte numérique des informations qui est **crypté** avec la clé **privée de l'autorité de certification**.

Ensuite :

- (1) Le certificat est conservé par le serveur (ce n'est pas forcément un serveur web ; d'autres services peuvent utiliser des certificats, mais on peut supposer que c'est un serveur web, pour simplifier).
- (2) Le certificat est présenté au client que se connecte au serveur (vous, avec votre navigateur).
- (3) Le client décrypte le condensat avec la clé publique de l'autorité de certification et récupère la clé publique du serveur.
- (4) Si tout va bien, il contacte l'autorité de certification pour vérifier que le certificat n'a pas été révoqué.

**Mais** d'où le client (vous) connaît-il la clé publique du serveur qui lui présente son certificat ? « *Les navigateurs web modernes intègrent nativement une liste de certificats provenant de différentes Autorités de Certification choisies selon des règles internes définies par les développeurs du navigateur* » : il peut donc y avoir un fonctionnement en cascade, une autorité de certification étant certifiée par une autre. Et bien sûr il y a un moment où il faut bien faire confiance.

La révocation par une autorité est possible, par exemple en cas de compromission. Remarquons aussi que les certificats ont une durée de vie limitée.

Quelques autorités de certification :

- La liste <https://webgate.ec.europa.eu/tl-browser/#/tl/FR> montre des autorités françaises qui sont soit publiques (Caisse des dépôts et consignations, Ministère de l'intérieur etc.), soit privées, certaines associées à des activités particulières (notaires, expert-comptables etc.) d'autre vendant simplement leurs services.
- **Lets'Encrypt** est une autorité de certification à but non lucratif : <https://letsencrypt.org/fr/>, financée, entre autres, par des fournisseurs de services et hébergeurs comme OVH en France.




# Appendices



## 1. Bibliothèques de calcul

(Retour sur l'exemple de la page 23)

À quelle vitesse peut calculer une machine contemporaine ? Résoudre un système linéaire de  $n$  équations à  $n$  inconnues est un bon test.

Regardons le programme python suivant (vous pouvez sauter directement à la ligne ):

```
1 import numpy as np
2 from scipy.linalg import lu_factor,lu_solve
3 import time
4 n=5000
5 A=np.random.rand(n,n)
6 B=np.random.rand(n)
7 #
8 c1=time.time()
9 lu, piv = lu_factor(A)
10 X= lu_solve((lu,piv),B)
11 c2=time.time()
12 #
13 n3= 2.*n**3/3.
14 t=c2-c1
15 gflops=n3/t/10**9
16 print("Gigaflops",gflops)
```

Passons sur les lignes 1 à 3 ; aux lignes 4,5 et 6 on fabrique un tableau au hasard de taille 5000 par 5000, et un second membre B de taille 5000. Les deux tableaux définissent le système d'équations. La résolution du système de 5000 équations est effectuée aux lignes 9 et 10. Comme ces lignes sont entourées de deux mesures de "l'heure", la différence (ligne 14) donne le temps calcul  $t$  en secondes.

 Le jeu commence ici :

- (1) On peut montrer que la résolution (par la méthode de Gauss, qui est à peu près celle qu'on apprend au collège) « coûte »  $\frac{2}{3}n^3$  opérations (additions et multiplications) pour un système de  $n$  équations, donc ici  $\frac{2}{3} \times 5000^3$  opérations soit  $\frac{2}{3} \times 125 \times 10^9$  opérations (environ 83,33... milliards d'opérations).
- (2) Sur ma machine (Intel i5, 3,5 Ghz, 4 cœurs), le temps calcul est de 0.779 secondes.

Donc, ma machine est capable d'effectuer  $\frac{2}{3} \times 125 \times 10^9 / 0.779$  opérations par seconde, soit environ **106 milliards d'opérations par seconde !** (on dit : 106,9 gigaflops).

Diab! Comment est-ce possible ?

- La machine tourne à 3.5 Gigahertz, ce qui permet à *chaque cœur* d'effectuer 3,5 milliards d'opérations par seconde.
- Mais, ma machine a 4 cœurs, et le programme est "multithreadé" (parallélisé sur les 4 cœurs), donc on peut atteindre  $4 \times 3.5 \times 10^9 = 14$  Gigaflops. Mais ça ne suffit pas à expliquer les 106,9 gigaflops observés.

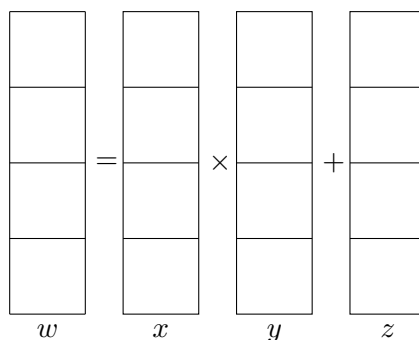


FIGURE 5. AVX (256)

- La réponse vient des instructions “avx” des processeurs Intel. Supposons qu’on ait 4 “vecteurs”  $w, x, y$  et  $z$  de taille  $4^{77}$  (voir la figure 5) ; alors, les instructions avx permettent de calculer  $w = x \times y + z$  en *un seul tour d’horloge*<sup>78</sup>, soit 8 opérations (4 additions et 4 multiplications) par tour d’horloge. Chaque cœur, quand il rencontre ce genre d’opération peut donc atteindre la vitesse de  $3,5 \times 8 = 28$  gigaflops (milliards d’opérations par seconde). Donc la vitesse maximale de ma machine (il y a 4 cœurs) est de  $4 \times 28 = 112$  gigaflops.

Comment est effectué le calcul dans mon programme ? En fait, les lignes 11 et 12 passent la main à la bibliothèque `lapack` (libre et optimisée pour chaque type de machine). Écrire des programmes qui permettent d’approcher d’aussi près les performances théoriques de la machine (106 pour 112) est difficile (c’est quasiment un métier en soi) : **il faut utiliser des bibliothèques !**

Remarque finale : il y a là deux types de parallélisme :

- (1) Les 4 cœurs font (peuvent faire) des calculs différents sur des données différentes : on parle de parallélisme **MIMD** (Multiple Instructions Multiple Data).
- (2) Les instructions “avx” : ont fait en parallèle le même calcul sur des données différentes ( $w_i = x_i \times y_i + z_i$ ). On parle de parallélisme **SIMD** (Single Instruction Multiple Data). C’est ce que font les GPUs<sup>79</sup> qui ont de très nombreux cœurs de calcul (7000 environ pour les plus puissantes), mais les cœurs font tous la même chose au même moment (idéal pour additionner deux tableaux, terme à terme) : si vous avez 7000 cœurs cadencés seulement à une vitesse de 1 Ghz (les GPUs ont des vitesses d’horloge relativement faibles), vous atteindrez donc la vitesses de  $7000 \times 10^9$  opérations par seconde, soit 7 téraflops (7000 milliards d’opérations par seconde).

Les derniers processeurs Intel ont, pour chaque cœur, *deux* unités avx de 512 bits (au lieu de 256) qui permettent de calculer sur des vecteurs de taille 8 au

77. C’est l’AVX 256 ( $256 = 4 \times 64$ ). Les derniers processeurs ont un AVX 512 (vecteurs de taille 8).

78. Pour être plus précis :

pour  $i = 1$  à  $4$  :  $w_i = x_i \times y_i + z_i$   
est effectué en parallèle.

79. Processeurs graphiques.

lieu de 4. La vitesse maximale *théorique* de chaque cœur est donc  $2 \times 16 = 32$  fois la fréquence d'horloge.

## 2. Le COW (Copy On Write)

L'idée de base est : copier c'est lent et ça coûte de la place. On ne va donc copier (un fichier ou toute autre chose) que quand c'est nécessaire. Par exemple, si deux utilisateurs (deux processus) accèdent à un fichier uniquement en lecture, pourquoi le copier ? En revanche si l'un des deux processus veut écrire sur le fichier, là il faut faire une copie pour donner à chacun son propre fichier. Comment implanter ça ? En voici une idée<sup>80</sup> :

- (1) Un premier processus accède au contenu du fichier par un pointeur qui montre au processus le « vrai » contenu du fichier (voir figure 6a). Le pointeur est un objet de petite taille, donc peu coûteux à copier.
- (2) Un deuxième processus accède aussi en lecture au même fichier (voir figure 6b). On crée un deuxième pointeur qui pointe sur les mêmes données. Mais en plus les données contiennent un compteur de références qui contient le nombre de pointeurs les désignant. Ce compteur qui valait 1 à l'origine vaut maintenant 2.
- (3) Imaginons qu'un des processus veuille écrire sur le fichier ; alors (voir figure 6c) :
  - (a) On en fait une copie (c'est là le « copy on write »).
  - (b) On a deux versions séparées, qui ont toutes les deux un compteur de références égal à 1.

L'effacement d'un fichier partagé en lecture par deux pointeurs est encore plus simple : si le premier processus veut effacer le fichier, on supprime son pointeur et on décrémente le compteur de références de 1. Un fichier (ses données) ne peut vraiment être effacé que quand le compteur de références est égal à 1.

### Applications :

- Les systèmes de « snapshots » (instantanés) des systèmes de fichiers **Zfs** et **Btrfs**. la technique est à peu près celle-ci :  
 Pour faire un snapshot de **/home** (par exemple), on crée un répertoire de nom **home-12h30**, par exemple ; on copie tous les pointeurs de **/home** dans **home-12h30**, ce qui peut être fait très rapidement<sup>81</sup>. Après cela, on a dans le nouveau répertoire **home-12h30** une *vue* de **/home** tel qu'il était quand on a fait le snapshot, et cela pour un « prix » très faible. **home** va continuer à évoluer et, si on ne touche pas à **home-12h30**, celui-ci va conserver l'état de **home** quand on a fait le *snapshot*.
- Cette technique d'instantanées est aussi implantée dans LVM sous Linux (voir [11]).
- Un « snapshot » n'est pas une sauvegarde ! Mais sauvegarder un « snapshot » permet de faire une sauvegarde d'un répertoire dans l'état exact où il était au début de la sauvegarde.
- **Btrfs** utilise aussi un système de « COW » pour remplacer le *journal* des systèmes de fichiers comme **ext4**.

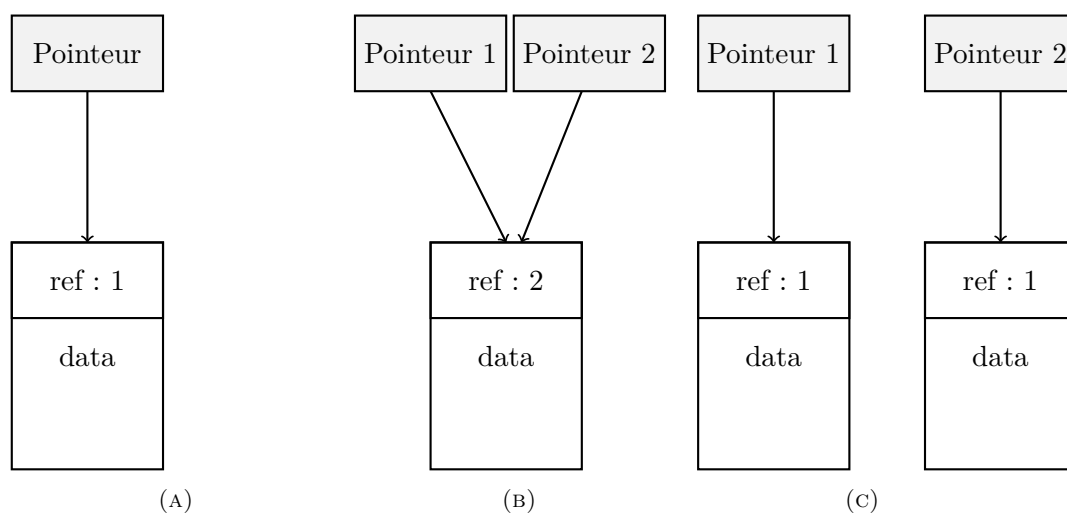


FIGURE 6. Copy On Write

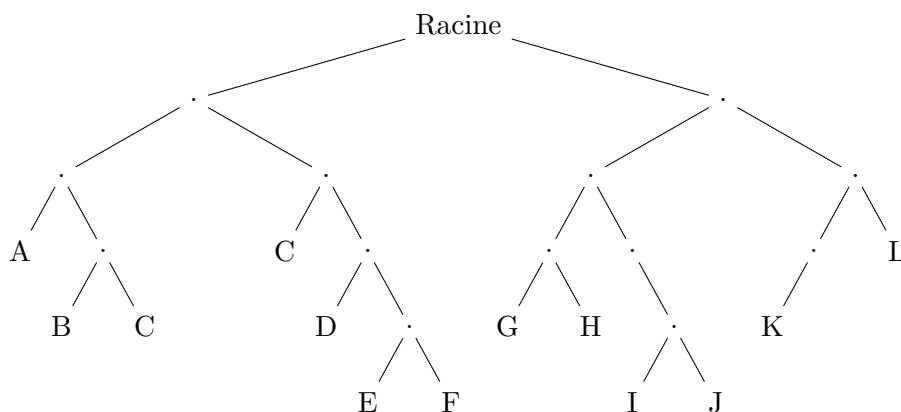


FIGURE 7. Un arbre binaire

### 3. Arbres binaires équilibrés (Balanced Trees, B-Trees)

Les *arbres binaires* sont une structure de données très efficace pour accéder rapidement à des données. Un arbre binaire (voir figure 7) est formé :

- de *nœuds* : depuis chaque *nœud* s'échappent zéro, une ou deux *branches*.
- Le *nœud* initial est appelé *racine*.
- Au bout de chaque branche, il y a un *nœud*. Si le *nœud* n'a pas de branche qui s'en échappe, on dit que c'est une *feuille*.

80. Ce n'est pas sans rapport avec les « liens hard », cf. page 25.

81. Il faut sûrement être un peu plus astucieux. Par exemple, appliquer la même technique... aux pointeurs ?

Le *niveau* d'un nœud ou d'une feuille est le nombre de branches qui faut parcourir pour l'atteindre +1.

Ainsi (figure 7) :

- *Racine* est au niveau 1.
- A, B, C, D, E, F, G, H, I, J, K, L, M sont respectivement aux niveaux : 4, 5, 5, 4, 5, 6, 6, 5, 5, 6, 6, 5, 4.

Supposons maintenant qu'on veuille ranger la liste : 50, 30, 40, 39, 42, 41, 20, 19, 22, 21, 25, 45, 43, 47, 70, 60, 58, 57, 62, 59, 80, 79, 65, 64, 67, 78, 90 dans un arbre binaire. On va :

- Installer 50 à la racine,
- Puis, 30 étant inférieur à 50 on l'installe au bout de la branche de gauche issue de 50.
- $40 < 50$  on va dans la branche de gauche issue de la racine, mais comme  $40 > 30$ , on l'installe dans la branche de droite issue de 30 (figure 8 (8a)).
- Bref : on part de la racine en allant à gauche ou à droite selon que le nombre est inférieur ou supérieur à la racine. Et chaque fois qu'on rencontre un nœud, on va à gauche ou à droite selon que la valeur à insérer est inférieure ou supérieure à celle du nœud. On insère la valeur quand une branche sélectionnée est vide (figure 8 (8b, 8c et 8d)).

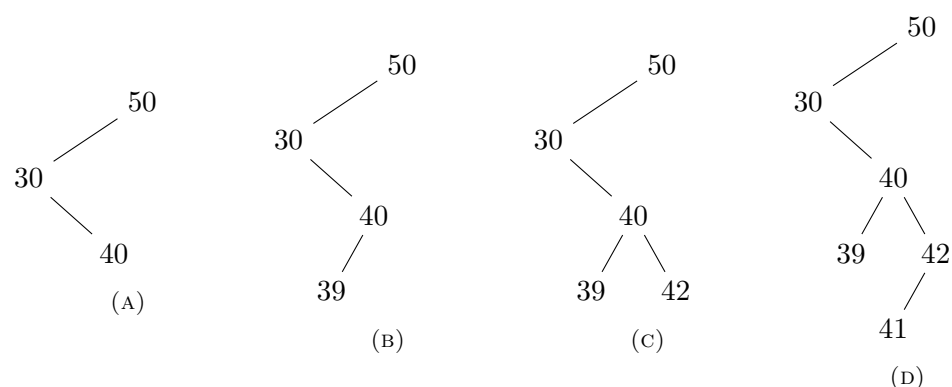


FIGURE 8. Insertions successives dans un arbre binaire

Évidemment, dans l'exemple donné ci-dessus on aurait pu remplacer les nombres par des chaînes de caractères, ou tout type d'objets qu'on peut comparer entre eux.

Considérons maintenant un arbre binaire de  $n$  niveaux ( $n = 2, 3, \dots$ ).

**C'est là que ça devient remarquable !** Pour retrouver un nombre dans l'arbre il faut parcourir au plus  $n$  niveaux, soit encore faire  $n$  comparaisons et  $n$  branchements au plus.

Mais combien y a-t-il de nœuds et donc *d'objets* (de nombres) rangés dans un arbre de  $n$  niveaux ? On suppose que l'arbre est complet (cf. figure 9), c'est à dire que tous les niveaux sont complètement occupés :

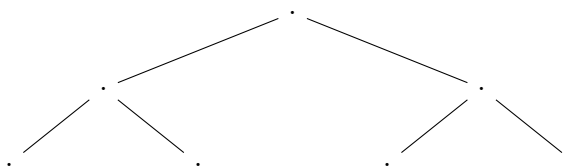


FIGURE 9. Arbre binaire complet (3 niveaux)

- au niveau 1, on a : 1 nœud.
- au niveau 2 : 2 nœuds, soit en tout  $2 + 1 = 3$  nœuds dans l'arbre, de la racine à ce niveau.
- au niveau 3 : 4 nœuds, soit en tout  $1 + 2 + 4 = 7$  nœuds jusqu'à ce niveau (à chaque niveau, le nombre de nœuds ajouté est le double de celui ajouté au niveau précédent).
- ...
- au niveau  $n$  : il y a  $2^{n-1}$  nœuds, ce qui fait qu'entre le niveau 1 et le niveau  $n$  (compris), on stocke  $1 + 2 + 2^2 + 3^2 + \dots + 2^{n-1}$  soit exactement  $2^n - 1$  nœuds<sup>82</sup>.

La recherche est *très rapide* :

- pour 1000 objets rangés dans un arbre binaire complet, comme  $2^{10} = 1024$ , les recherches se termineront en au plus 10 étapes. Dans un arbre binaire complet contenant un milliard d'objets, comme  $10^9 = 1000^3 < 1024^3 = 2^{10^3} = 2^{30}$ , les recherches se termineront en au plus 30 étapes.
- Supposons que la population mondiale est rangée dans un arbre binaire complet (par ordre de date de naissance ?) : comme il y a environ 6 milliards d'individus ( $6 \times 10^9$ ) on trouve qu'il faut au maximum 33 étapes pour trouver un individu !<sup>83</sup>

**3.1. Oui, mais...** Que se passe-t-il quand on insère une liste ordonnée dans un arbre binaire ? (voir la figure 10).

On engendre non pas un arbre binaire (toutes les branches gauches sont vides) mais une liste et la recherche à un coût qui n'est pas celui qu'on attend !

Cela vient du fait que le l'arbre n'est pas *équilibré* (*balanced*).

*Un arbre binaire est dit équilibré si les niveaux des feuilles diffèrent au plus de 1.*

Mais au prix de quelques manipulations, *on peut toujours s'arranger pour garder l'arbre équilibré au fur et à mesure des insertions*. Reprenons comme exemple l'insertion de la liste (1, 2, 3, 4, 5). Après l'insertion de (1) et (2) l'arbre est équilibré (deux branches, une de longueur 1 –la branche droite–, l'autre de longueur 0 –la branche gauche–). Mais à l'insertion de (3), l'arbre n'est plus équilibré (voir figure 11).

82. C'est le problème de l'échiquier de Sissa [13] : « En Inde, le roi Belkib (ou Bathait), qui s'ennuie à la cour, demande qu'on lui invente un jeu pour le distraire. Le sage Sissa invente alors un jeu d'échecs, ce qui ravit le roi. Pour remercier Sissa, le roi lui demande de choisir sa récompense, aussi fastueuse qu'elle puisse être. Sissa choisit de demander au roi de prendre le plateau du jeu et, sur la première case, poser un grain de riz, ensuite deux sur la deuxième, puis quatre sur la troisième, et ainsi de suite... ».

83. Le résultat est qu'une recherche parmi  $n$  objets coûte au maximum  $\log_2 n$  opérations ( $\log_2 2^p$ , c'est  $p$ ).



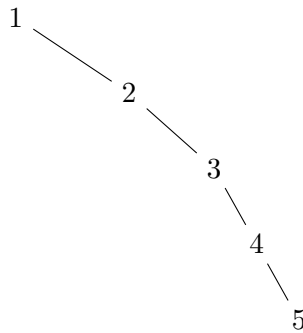


FIGURE 10. Insertion d'une liste ordonnée

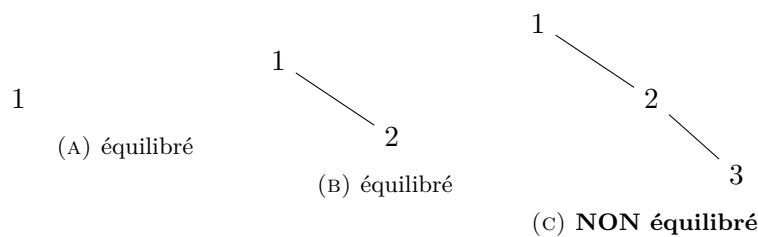


FIGURE 11. Insertion successive de 1, 2, 3

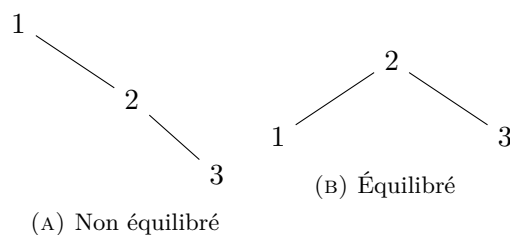


FIGURE 12. Équilibrage

Mais on peut rendre l'arbre équilibré par une transformation illustrée à la figure 12 : entre 12a et 12b on a fait remonter (2) d'un niveau, et du coup (1) vient à sa gauche, et l'arbre binaire est équilibré, et le coût d'une recherche est optimal.

#### 4. Une autre « source » de méthodes pour la cryptographie : le logarithme discret

On peut partir de quelque chose de simple (mais rien n'est difficile ici) : une horloge qui n'indique que les heures, numérotées 0, 1, 2, 3,...,10,11.

On sait bien que s'il est 10 heures, dans 4 heures il sera 2 heures. Plus exactement :

- l'addition est définie ainsi : si  $a$  et  $b$  sont des « heures », le résultat de  $a + b$  est le reste de la division de  $a + b$  par 12. ainsi  $10 + 4 = 14$  et le reste de 14 divisé par 12 est 2. Donc, dans notre arithmétique de l'horloge,  $10 + 4 = 2$ .
- On peut de la même manière définir la multiplication : si  $a$  et  $b$  sont deux nombres de l'intervalle  $0, \dots, 11$  (bornes incluses), alors on définit  $a \times b$  comme le reste de la division du produit de  $a$  par  $b$  divisé par 12. Exemples :
  - $5 \times 7$  est le reste de 35 divisé par 12, soit 11.
  - $4 \times 3 = 0$ .

On appelle ces nombres les *entiers modulo 12*.

Cela devient intéressant quand on remplace 12 par un nombre premier. Ainsi pour 7, donc pour les *entiers modulo 7*, on obtient les tables d'addition et de multiplication suivantes :

| + | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| 1 | 1 | 2 | 3 | 4 | 5 | 6 | 0 |
| 2 | 2 | 3 | 4 | 5 | 6 | 0 | 1 |
| 3 | 3 | 4 | 5 | 6 | 0 | 1 | 2 |
| 4 | 4 | 5 | 6 | 0 | 1 | 2 | 3 |
| 5 | 5 | 6 | 0 | 1 | 2 | 3 | 4 |
| 6 | 6 | 0 | 1 | 2 | 3 | 4 | 5 |

Table d'addition.

| * | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| 2 | 0 | 2 | 4 | 6 | 1 | 3 | 5 |
| 3 | 0 | 3 | 6 | 2 | 5 | 1 | 4 |
| 4 | 0 | 4 | 1 | 5 | 2 | 6 | 3 |
| 5 | 0 | 5 | 3 | 1 | 6 | 4 | 2 |
| 6 | 0 | 6 | 5 | 4 | 3 | 2 | 1 |

Table de multiplication.

et en supprimant les 0 pour la table de multiplication :

| * | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | 1 | 2 | 3 | 4 | 5 | 6 |
| 2 | 2 | 4 | 6 | 1 | 3 | 5 |
| 3 | 3 | 6 | 2 | 5 | 1 | 4 |
| 4 | 4 | 1 | 5 | 2 | 6 | 3 |
| 5 | 5 | 3 | 1 | 6 | 4 | 2 |
| 6 | 6 | 5 | 4 | 3 | 2 | 1 |

On peut facilement vérifier que :

- Par exemple :  $3^6 = 1$ ,  $3^2 = 2$ ,  $3^1 = 3$ ,  $3^4 = 4$ ,  $3^5 = 5$  et  $3^3 = 6$
- Les nombres  $3^p$  parcourent toutes les valeurs possibles (1, 2, ..., 6), et une seule fois. C'est vrai aussi pour 5. Mais ce n'est pas vrai pour 4, pour lequel les nombres  $4^p$  parcourent juste les valeurs 4, 2, et 1.

On dit que 3 et 5 sont des *générateurs*.

On choisit un générateur  $a$ . Le logarithme discret<sup>84</sup> à base  $a$  de  $x$  est le nombre  $n$  tel que  $a^n = x$ , ce qu'on écrit  $\log_a(x) = n$ . Par exemple, (voir ci-dessus)  $3^3 = 6$ , donc  $\log_3(6) = 3$  (sous-entendu : dans les entiers modulo 7).

- (1) Au lieu de calculer dans les *entiers modulo 7* comme ci-dessus, on choisit (au hasard) un très grand nombre premier  $P$  (120 chiffres, par exemple, on a déjà vu que c'était facile), quelque chose comme

<sup>84</sup>. sous-entendu : dans les entiers modulo 7.

08499652057794414184160279146447576156019630125837339050723288132709  
9544824336871492352319491041397008775809148491161129

et on calcule dans *les entiers modulo ce nombre* (on ne va pas écrire la table de multiplication!).

- (2) Il existe de bons algorithmes pour trouver des *générateurs*. On choisit un *générateur* qu'on appelle  $g$ .
- (3) On choisit un nombre  $x$  compris entre 1 et notre grand nombre premier  $P$ . C'est particulièrement facile. On calcule  $k = g^x$ . C'est facile.
- (4) La clé **publique**, c'est  $(g, k)$ .
- (5) Et la clé **privée** c'est  $x$ .

On a  $x = \log_g k$  :  $x$  est le logarithme discret de base  $g$  (sous entendu dans les entiers module  $P$ ) de  $k$ . **Et ce problème (le calcul du logarithme discret) est un problème à coût non polynomial, donc beaucoup trop coûteux pour être effectué.**



## Bibliographie

- [1] Rittaud B. *La nouvelle exponentielle de Nicolas Sarkozy*. URL : <https://mythesmanciesetmathematiques.wordpress.com/2016/08/23/la-nouvelle-exponentielle-de-nicolas-sarkozy/>.
- [2] HOW-TO-GEEK. *How to Use the ip Command on Linux*. URL : <https://www.howtogeek.com/657911/how-to-use-the-ip-command-on-linux/>.
- [3] PARI. *Pari/GP home*. URL : <http://pari.math.u-bordeaux.fr/>.
- [4] SAGE. *Sage*. URL : <https://www.sagemath.org/>.
- [5] Wiki UBUNTU. *Le système de fichiers BTRFS*. URL : <https://doc.ubuntu-fr.org/btrfs>.
- [6] WIKI-UBUNTU.FR. *Uncomplicated Firewall*. URL : <http://doc.ubuntu-fr.org/ufw>.
- [7] WIKIPÉDIA. *Domain Name System*. URL : [https://fr.wikipedia.org/wiki/Domain\\_Name\\_System](https://fr.wikipedia.org/wiki/Domain_Name_System).
- [8] WIKIPÉDIA. *Dynamic Host Configuration Protocol*. URL : [https://fr.wikipedia.org/wiki/Dynamic\\_Host\\_Configuration\\_Protocol](https://fr.wikipedia.org/wiki/Dynamic_Host_Configuration_Protocol).
- [9] WIKIPÉDIA. *IPv4*. URL : <https://fr.wikipedia.org/wiki/IPv4>.
- [10] WIKIPÉDIA. *IPv6*. URL : <https://fr.wikipedia.org/wiki/IPv6>.
- [11] WIKIPÉDIA. *Logical Volume Management*. URL : [https://fr.wikipedia.org/wiki/Gestion\\_par\\_volumes\\_logiques#Instantan%C3%A9s\\_\(snapshots\)](https://fr.wikipedia.org/wiki/Gestion_par_volumes_logiques#Instantan%C3%A9s_(snapshots)).
- [12] WIKIPÉDIA. *Observatoire Vera-C.-Rubin*. URL : [https://fr.wikipedia.org/wiki/Observatoire\\_Vera-C.-Rubin](https://fr.wikipedia.org/wiki/Observatoire_Vera-C.-Rubin).
- [13] WIKIPÉDIA. *Problème de l'échiquier de Sissa*. URL : [https://fr.wikipedia.org/wiki/Probl%C3%A8me\\_de\\_l%27%C3%A9chiquier\\_de\\_Sissa](https://fr.wikipedia.org/wiki/Probl%C3%A8me_de_l%27%C3%A9chiquier_de_Sissa).
- [14] WIKIPÉDIA. *Rémy Card*. URL : [https://fr.wikipedia.org/wiki/R%C3%A9my\\_Card](https://fr.wikipedia.org/wiki/R%C3%A9my_Card).
- [15] WIKIPEDIA. *Advanced Encryption Standard*. URL : [https://fr.wikipedia.org/wiki/Advanced\\_Encryption\\_Standard](https://fr.wikipedia.org/wiki/Advanced_Encryption_Standard).
- [16] WIKIPEDIA. *Algorithme de Karatsuba*. URL : [https://fr.wikipedia.org/wiki/Algorithme\\_de\\_Karatsuba](https://fr.wikipedia.org/wiki/Algorithme_de_Karatsuba).
- [17] WIKIPEDIA. *Autotools*. URL : <https://fr.wikipedia.org/wiki/Autotools>.
- [18] WIKIPEDIA. *Bibliothèque*. URL : [https://fr.wikipedia.org/wiki/Biblioth%C3%A8que\\_logicielle](https://fr.wikipedia.org/wiki/Biblioth%C3%A8que_logicielle).
- [19] WIKIPEDIA. *Chiffrement RSA*. URL : [https://fr.wikipedia.org/wiki/Chiffrement\\_RSA](https://fr.wikipedia.org/wiki/Chiffrement_RSA).
- [20] WIKIPEDIA. *Cmake*. URL : <https://fr.wikipedia.org/wiki/CMake>.
- [21] WIKIPEDIA. *Datagramme*. URL : <https://fr.wikipedia.org/wiki/Datagramme>.
- [22] WIKIPEDIA. *File system*. URL : [https://fr.wikipedia.org/wiki/Syst%C3%A8me\\_de\\_fichiers](https://fr.wikipedia.org/wiki/Syst%C3%A8me_de_fichiers).
- [23] WIKIPEDIA. *Free desktop*. URL : <https://fr.wikipedia.org/wiki/Freedesktop.org>.
- [24] WIKIPEDIA. *Git*. URL : <https://fr.wikipedia.org/wiki/Git>.
- [25] WIKIPEDIA. *Internet Control Message Protocol*. URL : [https://fr.wikipedia.org/wiki/Internet\\_Control\\_Message\\_Protocol](https://fr.wikipedia.org/wiki/Internet_Control_Message_Protocol).
- [26] WIKIPEDIA. *iptables*. URL : <https://fr.wikipedia.org/wiki/Iptables>.
- [27] WIKIPEDIA. *Louis Pouzin*. URL : [https://fr.wikipedia.org/wiki/Louis\\_Pouzin](https://fr.wikipedia.org/wiki/Louis_Pouzin).

- [28] WIKIPEDIA. *make*. URL : <https://fr.wikipedia.org/wiki/Make>.
- [29] WIKIPEDIA. *Modèle OSI*. URL : [https://fr.wikipedia.org/wiki/Mod%C3%A8le\\_OSI](https://fr.wikipedia.org/wiki/Mod%C3%A8le_OSI).
- [30] WIKIPEDIA. *Netfilter*. URL : <https://fr.wikipedia.org/wiki/Netfilter>.
- [31] WIKIPEDIA. *Nombre RSA*. URL : [https://fr.wikipedia.org/wiki/Nombre\\_RSA#RSA-240](https://fr.wikipedia.org/wiki/Nombre_RSA#RSA-240).
- [32] WIKIPEDIA. *SHA-2*. URL : <https://fr.wikipedia.org/wiki/SHA-2>.
- [33] WIKIPEDIA. *Système de fichiers journalisé*. URL : [https://fr.wikipedia.org/wiki/Syst%C3%A8me\\_de\\_fichiers\\_journalis%C3%A9](https://fr.wikipedia.org/wiki/Syst%C3%A8me_de_fichiers_journalis%C3%A9).
- [34] WIKIPEDIA. *Transmission Control Protocol*. URL : [https://fr.wikipedia.org/wiki/Transmission\\_Control\\_Protocol](https://fr.wikipedia.org/wiki/Transmission_Control_Protocol).
- [35] WIKIPEDIA. *User Datagram Protocol*. URL : [https://fr.wikipedia.org/wiki/User\\_Datagram\\_Protocol](https://fr.wikipedia.org/wiki/User_Datagram_Protocol).