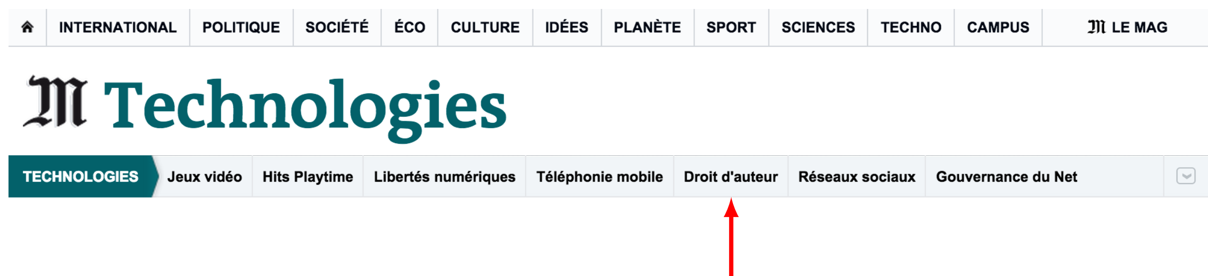
Piratage #1 : Peut-on empêcher le partage de fichiers ?

Charles Bouillaguet

26 janvier 2018

La question du *droit d'auteur*, ou plus généralement, de la *propriété intellectuelle* agite régulièrement les entreprises qui agissent sur internet, et même plus largement le milieu des gens qui s'intéressent plus ou moins à ce qui se passe sur la toile. À témoin, on prendra le site internet du journal *Le Monde* :



En effet, le « problème », si on ose dire, avec le développement de l'informatique en général, et d'internet en particulier, c'est que copier un fichier est très facile. Du coup, la duplication non-autorisée (« pirate ») d'oeuvres intellectuelles (programmes, livres, artices scientifiques, films, morceaux de musiques, etc.) s'est répandue et généralisée.

Cela a commencé en 1999 avec `Napster`. Après avoir atteint 25 millions d'utilisateurs uniques en janvier 2001, ce réseau P2P de partage de fichiers (essentiellement musicaux) a été fermé par les tribunaux en juillet 2001.

D'autres protocoles (et implémentations logicielles associées) ont succédé à Napster : FastTrack (Kazaa), eDonkey (eMule), etc.

Depuis 2007, il semble que le protocole BitTorrent domine le monde du partage de fichiers.

1 Le protocole BitTorrent

L'avantage du protocole BitTorrent est d'être particulièrement simple.

Quand deux pairs établissent une connection, il y a une phase de *handshake*, dans laquelle chaque pair envoie :

- Un en-tête fixé
- Un *hash* qui identifie le « torrent »
- Un *hash* qui identifie le pair

Ensuite, les deux pairs peuvent s'envoyer l'un des 8 types de messages prévus par le protocole :

`choke`	je ne t'envoie plus de données
`unchoke`	je veux bien te ré-envoyer des données
`interested`	je veux des données que tu as
`not interested`	tu n'as rien que je veuille
`have`	je viens de recevoir le morceau n°i
`bitfield`	voici la liste des morceaux que j'ai
`request`	je voudrais le morceau n°i
`piece`	voici le morceau n°i
`cancel`	arrête de m'envoyer le morceau n°i

L'idée du mécanisme de « choking » est multiple. D'une part il permet de réguler le débit. D'autre part, il permet d'imposer une forme de réciprocité : si tu ne m'envoies rien, ou bien avec un débit trop faible, je te `choke`.

L'intérêt de `interested`, c'est que ça permet de savoir si un pair qui est pour l'instant `choked` aurait envie de reprendre le transfert si on le `unchoke`-ait.

L'implémentation « standard » du client BitTorrent `unchoke` les 4 pairs les plus rapides, et tente périodiquement d'en `unchoker` d'autres, au cas où ils se réveleraient être mieux (c'est le « optimistic unchoking »).

Il reste un « petit détail » à régler : comment trouve-t-on l'adresse IP des pairs qui sont intéressés ? Un mécanisme de « Peer Exchange » a été conçu ultérieurement, grace auquel les pairs peuvent échanger des informations sur les pairs qu'ils connaissent (en s'envoyant des listes de pairs actifs ou inactifs).

Mais ceci ne règle pas la question du *bootstraping* : comment entre-t-on en contact avec « l'essaim » ? Dans le protocole BitTorrent, la solution apportée à ce problème est le *tracker* : c'est un mini-serveur web, auquel on accède via une requête `HTTP GET`, en lui envoyant le *hash* du torrent auquel on veut participer, le *hash* de notre identité, et de vagues informations sur notre état actuel (quelle quantité téléchargée, etc.).

Le tracker enregistre notre IP, et nous ajoute dans sa base de donnée de participants à l'essaim. Il nous renvoie en échange une liste de quelques autres participants avec lequel on peut initier des échanges.

2 Centralisme ou anarchie ?

Le tracker est évidemment le point faible du système, tout comme les moteurs de recherche dédiés aux fichiers `.torrent`. En effet, les deux types de serveur sont périodiquement menacés de poursuites légales, et ferment les uns après les autres. C'est essentiellement le même problème qui a conduit à la disparition des autres systèmes de P2P. Le problème est le caractère *centralisé* du système : il y a un point de faiblesse (« *single point of failure* »).

Quand on y réfléchit bien, une bonne partie des protocoles importants d'internet sont centralisés (le système DNS, par exemple, ce qui permet à certains états de le manipuler pour bloquer ou compliquer l'accès à certains sites).

La question qui se pose est donc : est-il possible de concevoir un système de partage de fichier *complètement décentralisé* ? Un système « anarchique » (pas d'autorité centrale, en mode « ni dieu ni maître »), sans chef d'orchestre, sans serveur central ? L'intérêt d'un tel système est entre autre la robustesse (au moins face aux problèmes « legaux ») : si aucun participant n'a un rôle spécifique qui le distingue des autres, et fasse de lui une cible, on peut espérer que tous soient un peu protégés par leur nombre.

En fait, ce qu'il nous faut, c'est une *table de hachage distribuée* (DHT). C'est-à-dire qu'il faut qu'on puisse stocker (et rechercher) des paires clef-valeurs de façon décentralisée. Ce serait suffisant pour implémenter un *tracker réparti*, par exemple : le *hash* d'un torrent peut faire office de clef, et la liste des identifiants des participants au nuage peut faire office de valeur. Ca peut aussi servir à implémenter un moteur de recherche décentralisé.

Une telle table de hachage distribuée/répartie suppose que chacun des noeuds qui participe au système ne stocke pas *toutes* les paires clef-valeur, mais juste une partie. En plus, il n'est pas envisageable que chaque noeud « sache » *où se trouve* la valeur associée à une clef particulière.

Pour cela, il faudra résoudre un certain nombre de problèmes : comment déterminer qui possède la valeur associée à une clef donnée (localisation de ressource) ? Ou, dans l'autre sens : comment affecter une valeur à une clef donnée ? Et comment s'organiser pour "résister" si un noeud, ou un groupe de noeuds, se déconnecte brutalement ?

3 Les « petits mondes » : l'informatique imite la société humaine

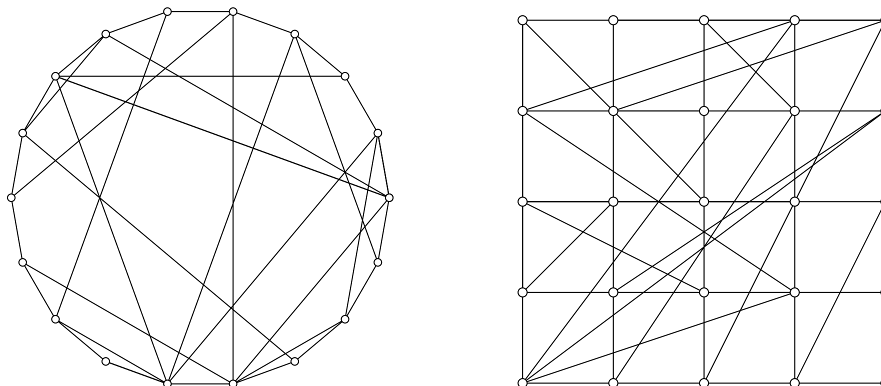
On s'est tous dit un jour « oh, mais le monde est petit ! » Vers la fin des années 1960, un groupe de chercheurs tente de mesurer le phénomène. Dans une expérience menée en 1967 et demeurée célèbre, Stanley

Milgram (1933–1984) a proposé à des gens choisis au hasard dans le Kansas et le Nebraska d'acheminer des lettres vers une destination située dans le Massachusetts. Les points de départ et d'arrivée avaient été choisi parce qu'on les imaginait éloignés géographiquement et sociologiquement. Les « facteurs » devaient uniquement transmettre (éventuellement par courrier) la lettre à des gens qu'ils *connaissent*. On considère qu'on connaît quelqu'un quand on l'appelle par son prénom.

Sur les 296 lettres qui sont parties, 64 sont arrivées. La plupart étaient passées par 5 ou 6 intermédiaires. Milgram en a conclut qu'un citoyen américain connaissait tous les autres via 6 intermédiaires. Ceci se généralise sûrement à l'ensemble de l'humanité.

La conclusion de cette expérience est un peu surprenante. On a cherché à se l'expliquer. Pour cela on a chercher à modéliser les relations sociales par un graphe. Ce que l'expérience révèle, c'est qu'entre deux sommets un peu arbitraire de ce graphe, il existe un chemin *court* (de taille 6 pour un graphe de 300 millions de noeuds, en gros). En plus, le graphe n'est pas fait n'importe comment : si je connais Alice et que je connais Bob, alors Alice et Bob ont beaucoup plus de chance de se connaitre aussi que deux personnes prises au hasard dans le monde.

Aussi, les recherches se sont concentrées sur des graphes qui ressemblent à ça :



L'idée est qu'il y a une structure *régulière* (le cercle, la grille) qui modélise les liens usuels que les gens ont entre eux (cercle d'amis, cellule familiale, copains de classe, collègues, etc.). Et en plus, il y a un nombre, plus faible, de liens *irréguliers* (un par personne, ici), qui modélisent, eux, le hasard des rencontres (ici, chaque personne établi un lien complètement aléatoire avec une autre personne).

Il se trouve que ces graphes sont des graphes de « petit monde », c'est-à-dire qu'il y a toujours des chemins « courts » entre deux personnes. En l'occurence, « court » signifie « grosso-modo logarithmique en le nombre de noeuds du graphe ».

Application aux réseaux peer-to-peer Les graphes de petits mondes, étudiés d'abord dans le cadre de la sociologie, puis des « mathématiques amusantes », sont d'un intérêt particulier pour la construction de réseau P2P. En effet, dans un réseau P2P, les pairs établissent des connections entre eux, et on peut se représenter la chose par un graphe.

Premièrement, il est avantageux de n'être connecté qu'à un petit nombre de pairs. En effet, lorsqu'on entre dans ou qu'on sort de l'essaim, cela risque de nécessiter une action de la part de tous ceux avec qui ont est connecté. On ne peut pas envoyer un message à tout l'internet ! Par exemple, dans le protocole BitTorrent, on envoie un message `Have` à tous les pairs avec lesquels ont est connecté dès qu'on termine la réception d'une pièce. On ne peut donc pas être connecté à trop de pairs. On veut que le *degré* du graphe (c.a.d. le nombre maximal de voisin d'un noeud) soit faible.

Deuxièmement, si on veut implémenter une table de hachage distribuée, on a tout intérêt à ce que deux noeuds quelconques soient « proches ». En effet, si on cherche la valeur associée à la clef k, on peut demander à un de nos voisins de trouver k pour nous. Celui-ci peut lui-même demander à un de ses voisins, etc. S'il y a toujours des chemins courts, alors le processus sera rapide. C'est la même chose que l'acheminement des lettres de Milgram. On veut que le *diamètre* du graphe (c.a.d. le plus grande distance qui sépare deux noeuds, sachant que leur distance est la longueur du plus court chemin entre eux) soit faible.

Faible degré, faible diamètre = graphe des relations sociales, pour résumer.

Alors, il y a malheureusement des limites à ce qu'il est possible d'obtenir. Edward F. Moore (1925–2003)

a montré autour de 1960 qu'un graphe (connexe) de degré d et de diamètre k possède au maximum

$$1 + d\sum_{i=0}^{k-1}(d-1)^i = 1 + d\frac{(d-1)^k - 1}{d-2}$$

sommets. En effet, partons d'un sommet et faisons un parcours en largeur. Au niveau zéro, il y a le sommet de départ. Au niveau 1, il y a ses d voisins (au maximum). Au niveau 2, il y a (au maximum) les $(d-1)$ autres voisins des premiers. Au niveau i, il y a les $d(d-1)^i$ voisins de ceux de niveau $(i-1)$. Au niveau k on couvre tout le graphe par définition de k.

Il s'ensuit que si le nombre de connections établies par chaque noeud est borné par une constante, alors le diamètre doit être au moins logarithmique (mais ce n'est pas si mal).

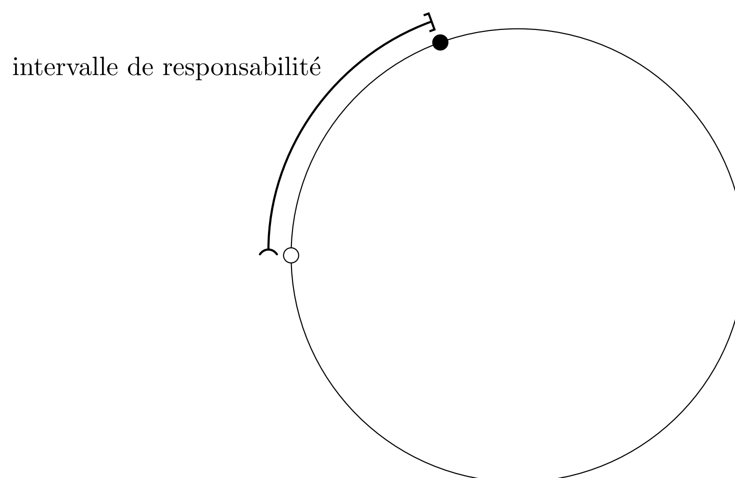
En fait, l'expérience de Milgram ne met pas seulement en lumière l'existence de chemins courts dans le graphe des relations sociales. Elle montre aussi que les gens sont assez bon pour *trouver* ces chemins courts, et ce de manière décentralisée. Il ne s'agit plus seulement de propriétés mathématiques, mais d'un problème *algorithmique*.

4 Chord : l'un des systèmes les plus simples

Construire des tables de hachage distribuées sur des graphes de petits monde n'est pas horriblement difficile, au moins en théorie (en pratique, c'est une autre affaire). L'un des premiers exemples, et des plus simples, est Chord. Il date de 2001.

L'idée maîtresse est la suivante : les clefs de la table de hachage sont des chaines de m bits. Par exemple, $m = 256$ avec une fonction de hachage de bonne qualité telle que SHA256, qui produit des empreintes de 256 bits. Par ailleurs, chaque noeud qui participe à la table possède une *identité* (un *Node ID*) sur m bits aussi.

Les noeuds sont organisés en anneau (on considère leur identifiant modulo 2^n) : mon successeur est le noeud qui a le numéro supérieur le plus proche. Chaque clef est *affectée* au noeud qui est son successeur immédiat :



L'intérêt de cette manière de faire, c'est que tant que la fonction de hachage répartit bien les clefs, chaque noeud se retrouve à devoir gérer une fraction environ $1/N$ des clefs, où N désigne le nombre de noeuds. Cette technique est également utilisée dans des bases de données NoSQL distribuées bien connues. De plus, lorsqu'un nouveau noeud arrive dans le système, on peut espérer que seule une fraction environ $1/N$ des clefs « change de main ».

4.1 Localisation d'une ressource

Comment récupérer la valeur associée à une clef dans le système ? Il faut en fait localiser le noeud reponsable de cette clef : c'est lui qui possède la valeur associée. Il faut donc être capable, étant donné un identifiant, de trouver son successeur (c'est lui qui est responsable). Pour cela, il faut que chaque noeud soit en permanence capable d'identifier le noeud qui est successeur. Ceci permet de localiser une paire clef-valeur : si nous n'en sommes pas responsable, c'est qu'elle est plus loin dans l'anneau ; on peut alors

s'adresser à notre successeur et lui refiler la mission de localiser la clef. De proche en proche, on va finir par atteindre la paire clef-valeur qu'on recherche (mais ça risque de prendre très longtemps).

Pour accélerer la procédure, il faut pouvoir prendre des raccourcis. Pour cela, chaque noeud (d'adresse n) maintien un tableau de m *fingers*. Le i-ème *finger* est le successeur de $n+2^i$. L'ensemble du dispositif tire son nom du fait que ces *fingers* tracent des liens qui sont des « cordes » du cercle.

Pour avancer le plus vite possible le long du cercle, il faut emprunter ces cordes, mais sans dépasser la destination :

```
def is_between(c, a, b):
"""
Renvoie True si en partant de a et en avançant dans l'anneau
on trouve sur c avant de tomber sur b (c est donc entre a et b).
"""
pass          # code laissé en exercice au lecteur.


class Node:
    successor = None
    predecessor = None

    def find_successor(self, ID):
        """
        La fonction critique. Permet de localiser le noeud qui
        contient la valeur associée à la clef ID.
        """
        p = self.find_predecessor(ID)
        return p.successor

    def find_predecessor(self, ID):
        p = self
        while not ID.is_between(p, p.successor):
            # Ceci est un Remote Procedure Call. On confie au noeud p
            # le soin de trouver la solution pour nous.
            p = p.closest_preceding_finger(ID)
        return p

    def closest_preceding_finger(self, ID):
        for finger in reversed(self.fingers):
            if is_between(finger, self, ID):
                return finger
        return self
```

Il faut imaginer que tout ceci s'implémente avec des Remote Procedure Calls (RPC), et donc que le fait d'invoquer une méthode sur un autre noeud effectue le calcul... sur une autre machine, puis rapatrie le résultat. La librairie standard de python contient un module, xmlrpc qui permet potentiellement d'implanter ça sans (trop de) douleur.

La boucle **while** de `find_predecessor` effectue au pire m itérations. En effet, à chaque fois qu'on saute au *finger* le plus éloigné mais valable, on divise au moins la distance par deux. En fait, on peut même dire un peu mieux : après i sauts, la distance restante entre le noeud courant et la cible est au plus $2^m/2^i$. Que se passe-t-il au bout de $\log_2 N$ sauts ? On se retrouve à distance $2^m/N$. Or, dans un intervalle de cette longueur, il y a (en moyenne) un seul noeud. On peut donc espérer que le processus s'arrête rapidement.

Au passage, amusons-nous à effectuer le processus de routage depuis tous les noeuds vers un identifiant donné *id*. Cela forme une collection de chemins qui converge vers *id*. Si on les « fusionne », cela donne un arbre *couvrant*, dont les arêtes pointent vers la racine. Plus les noeuds sont proches de la racine (de *id*), plus ils voient souvent passer des requêtes destinées à *id*. Cela peut donc valoir le coup de maintenir un petit cache afin de pouvoir répondre à la place de *id*, afin de répartir la charge.

4.2 Incorporation de nouveaux participants

Comment est-ce qu'un nouveau participant peut rejoindre l'anneau ? Il y a plusieurs manières de faire, qui varient dans leur efficacité, et leur capacité à gérer des situations bizarres (exemple : deux noeuds arrivent en même temps, ne se connaissent pas, or l'un devrait être le successeur de l'autre). On présente ici la méthode réputée la plus robuste.

Pour commencer, il lui faut être capable de contacter au moins un noeud de l'anneau. Ensuite, le nouveau participant détermine son propre identifiant (le haché de son adresse MAC, ou n'importe quoi d'autre). Enfin, il faut qu'il détermine sa propre position dans l'anneau et qu'il établisse les liens qu'il devrait avoir avec ses voisins. Pour cela, il effectue une recherche sur son propre identifiant, et grace à ceci il va découvrir le noeud qui est immédiatement *après* lui : ce sera son successeur.

```
# suite de la classe Node:
def join(self, other):
    """other = n'importe quel noeud."""
    self.predecessor = None
    # Ceci est un Remote Procedure Call. On demande à other de
    # déterminer qui est "responsable" de notre ID, c'est-à-dire qui
    # est notre successeur immédiat.
    self.successor = other.find_successor(self)

def stabilize(self):
    """ tout le monde doit appeler cette méthode périodiquement"""
    x = self.successor.predecessor       # RPC (propriété d'un noeud distant)
    if x != self:
        # quelqu'un s'est incrusté entre nous et notre successeur
        self.successor = x
    x.notify(self)                       # RPC. On rend visite au voisin d'après.

def notify(self, other):
    """ other nous dit que, d'après lui, il est notre prédécesseur"""
    if self.predecessor != None and is_between(self.predecessor, other, self):
        # il est plus loin que notre prédécesseur actuel : on l'ignore
        return
    self.predecessor = other

def refresh_fingers(self):
    """tout le monde doit appeler cette méthode périodiquement"""
    i = random.randrange(1, m)
    self.fingers[i] = self.find_successor(self.id + 2**i)
```

L'important pour raisonner sur la correction de ces procedures est que les fingers peuvent être complètement faux, ce n'est pas très grave. Au pire, ça *ralentit* l'opération de recherche. C'est pour ça que la mise à jour des *fingers* peut se faire de manière relaxée et asynchrone.

Par contre, il est INDISPENSABLE que le successeur (c.a.d. le finger n°0) soit correct. Tant que c'est le cas, l'opération de recherche renvoie toujours les bons résultats. Ensuite, on peut se convaincre que les deux propriétés suivantes sont vraies :

1. Une fois qu'un noeud est capable de « résoudre » un identifiant donné, alors il reste capable de le faire quoi qu'il arrive.
2. Au bout d'un certain temps après la dernière arrivée, tous les successeurs sont corrects.

Ces deux propriétés se démontrent en gros par récurrence. *Primo*, la procédure de stabilisation finit par s'arrêter. En effet, au fur-et-à-mesure des appels à `stabilize` et à `notify`, chaque noeud adopte un successeurs de plus en plus proche, or comme il n'y a qu'un nombre fini d'identifiants, ceci ne peut pas continuer à l'infini.

Secundo, il faut montrer que n'importe quel noeud est accessible depuis son prédecesseur, s'il en a un. Pour les noeuds qui arrivent dans l'anneau, c'est facile car ils n'en ont pas. Ensuite, le seul moment où le prédécesseur peut être modifié, c'est dans `notify`. Mais cette fonction ne peut être appelée que par un noeud capable de nous atteindre !

Tertio, il suffit de montrer qu'une fois qu'un noeud u peut atteindre un noeud v en suivant les pointeurs `successeur`, alors il peut toujours le faire. Le seul moment où les successeurs sont modifiés, c'est dans `stabilize`. On vient de voir que `self.successor` est toujours accessible depuis `self.successor.predecessor`. Par conséquent, même si `self.successor` est modifié, l'ancien noeud est toujours accessible, et donc tous ceux qui étaient accessible à partir de là le sont toujours.

On peut aussi se convaincre qu'une fois que le processus de stabilisation s'arrête (plus rien ne change) alors tous les noeuds ont des successeurs différents (sinon, un nouveau tour de `stabilize` changerait des choses).

Une fois que les choses semblent stabilisées, on peut envisager de transférer les clefs aux nouveaux arrivants.

4.3 Tolérance aux pannes

Comment faire pour survivre à la disparition ou à la déconnection subite d'un noeud ? Cela « coupe » le cercle et risque de tout casser...

De manière générale, la tolérance aux pannes dans ce genre de situation est invariablement atteinte grâce à une forme ou une autre de *redondance* et/ou de *replication* (et c'est là que les problèmes pratiques commencent, car dès qu'il y a plusieurs versions des mêmes données, il faut maintenir leur synchronisation).

Pour empêcher le départ d'un noeud de poser des problèmes trop graves, il suffit que chaque noeud garde en mémoire non pas *son* unique successeur, mais la liste de *ses* quelques successeurs. Si le prochain, ou même les deux prochains, disparaissent, on peut espérer s'en tirer. De la même manière, on peut garder une copie des données qui sont sous la responsabilité de notre successeur. S'il disparait, on en aura encore une version.

Cependant, à moins qu'il nous envoie des données de synchronisation, on a pas forcément la version la plus à jour (parce qu'une mise à jour peut nous passer « par-dessus »).

Ce que ces notes contiendront à l'avenir...

- La table de hachage distribuée Kademlia (utilisée dans bittorrent).
- Le protocole Tor.