Avalanche : Téléchargement peer-to-peer de fichier par combinaisons linéaires de blocs

Sujet proposé par Laurent Viennot
`http://gyroweb.inria.fr/~viennot/`

Difficulté : moyen (**) ou difficile (***) avec l'extension réseau.

Page de suivi et challenges :
`http://gyroweb.inria.fr/~viennot/enseignement/projets/avalanche/`

1 Introduction

Le téléchargement de pair à pair (ou « *peer-to-peer* » en anglais) de fichier consiste à utiliser la bande-passante des clients pour servir efficacement un grand nombre de clients. Le fichier est ainsi découpé en n blocs, ce qui permet aux clients en cours de téléchargement de s'échanger des blocs directement sans passer par la source. Ce principe d'échange est par exemple à la base du protocole BitTorrent [1] qui l'utilise pour inciter les clients à donner une partie de leur bande-passante : un client donne de préférence à ceux qui lui donnent, c'est le principe du « prêté pour un rendu » (ou « *tit for tat* » en anglais).

Cependant, quand le nombre de clients augmente la décision de télécharger tel ou tel bloc depuis tel ou tel client devient difficilement optimisable. BitTorrent utilise une heuristique consistant à essayer d'obtenir le bloc qui paraît le plus rare. Cependant des situation de blocage ou de ralentissement peuvent être observés pour certains clients. Par exemple, un client qui ne possède que des blocs possédés par tous les autres clients n'intéressera personne et obtiendra une vitesse de téléchargement faible.

Pour lever ce frein, le protocole Avalanche [2] propose d'échanger des combinaisons linéaires de blocs. Chaque combinaison possédant des informations provenant de tous les blocs est ainsi susceptible d'intéresser tout client qui n'a pas fini son téléchargement : il suffit de télécharger n combinaisons indépendantes pour être capable de reconstruire le fichier.

Le présent sujet propose de programmer la partie concernant le calcul des combinaisons linéaires, la vérification de leur intégrité et la reconstruction du fichier. Même si la compréhension du sujet nécessite d'introduire quelques notions sur le corps des entiers modulo un grand nombre premier (dont les opérations sont fournies dans les librairies de java), la réalisation repose sur des notions élémentaires d'algèbre linéaire.

En extension, on pourra traiter la partie réseau du client similaire à BitTorrent qui consiste à se connecter à d'autres clients et à s'échanger des données.

2 Description du protocole

Un fichier F est découpé en n blocs $B_1, \ldots, B_n$. Chaque bloc B_i est découpé en k nombres entiers positifs $B_i = (x_{i1}, \ldots, x_{ik})$. Pour effectuer les opérations de combinaison, on se donne un grand nombre premier q de plusieurs centaines de bits (chaque x_{ij} est supposé strictement inférieur à q). La source fournit des combinaisons aléatoires $a_1B_1 + \cdots + a_nB_n = (a_1x_{11} + \cdots + a_nx_{n1}, \ldots, a_1x_{1k} + \cdots + a_nx_{nk}) \, mod \, q$: les a_j sont tirés aléatoirement et les opérations sont effectuées modulo q. (On travaille bien dans un corps puisque q est premier.) Les coefficients a_j sont transmis en sus de la combinaison, induisant un certain sur-coût de communication.

2.1 Vérification des combinaisons

Une des difficulté du protocole réside dans la vérification de l'intégrité d'une combinaison : connaissant $a_1, \ldots a_n$, comment vérifier que la combinaison linéaire $C = (y_1, \ldots y_k)$ qu'on a reçu est bien égale à $a_1B_1 \cdots a_nB_n$? (Il faut pouvoir se protéger d'un client malicieux ou bogué qui introduit de mauvaises combinaisons.) Pour cela, le protocole propose d'utiliser une fonction de hachage h homomorphe, c'est-à-dire telle que $h(a_1B_1 + \cdots + a_nB_n) = h(B_1)^{a_1} \cdots h(B_n)^{a_n}$. Ainsi la connaissance de $h(B_1), \ldots, h(B_n)$ suffit pour vérifier l'intégrité de toute combinaison.

On utilisera ainsi la fonction de hachage obtenue en trouvant deux nombre premiers p et q tels que $p-1 = dq$ (q divise $p-1$), en se donnant k nombres aléatoires $g_1, \ldots, g_k$ et en posant :

$$h(y_1, \ldots y_k) = g_1^{dy_1} \cdots g_k^{dy_k} \, mod \, p$$

(On prendra pour les g_i des nombres aléatoires de même longueur que p.) La propriété d'homomorphisme découle alors du petit théorème de Fermat qui stipule $g^{p-1} = 1 \, mod \, p$ pour tous $g \neq 0$ et p est premier. Comme p et q sont choisis tels que $p - 1 = dq$, si $y_j = a_1x_{1j} + \cdots + a_nx_{nj} \, mod \, q$, on a $dy_j = d(a_1x_{1j} + \cdots + a_nx_{nj}) + m(p-1)$ pour un certain entier m. Comme $g_j^{m(p-1)} = 1 \, mod \, p$, $g_j^{dy_j} = g_j^{d(a_1x_{1j}+\cdots+a_nx_{nj})} \, mod \, p$, soit $g_j^{dy_j} = \left(g_j^{dx_{1j}}\right)^{a_1} \cdots \left(g_j^{dx_{nj}}\right)^{a_n} \, mod \, p$. On a donc bien $h(a_1B_1 + \cdots + a_nB_n) = h(B_1)^{a_1} \cdots h(B_n)^{a_n} \, mod \, p$.

La sécurité de cette fonction de hachage repose sur la difficulté de calculer des logarithmes discrets. Il est par exemple facile de produire de faux blocs d'empreinte donnée connaissant i, j, x, y tels que $g_i^{dx} = g_j^{dy} \, mod \, p$. Une description plus complète de la fonction de hachage ci-dessus est décrite dans [3].

2.2 Reconstruction du fichier

Le protocole repose sur le fait que n combinaisons aléatoires ont peu de chances d'être linéairement dépendantes. Étant donné n combinaisons $C_1, \ldots C_n$ supposées indépendantes, on reconstruira le fichier en utilisant la méthode du pivot de Gauss. En effet, le problème revient à résoudre un système linaire du type $AB = C$: notons $a_{i1}, \ldots, a_{in}$ les coefficients de la combinaison $C_i = a_{i1}B_1 \cdots + a_{in}B_n = (y_{i1}, \ldots, y_{in}) \, mod \, q$. Chaque vecteur $B'_j = (x_{1j}, \ldots, x_{nj})$ est alors solution du système $AB'_j = C'_j$ avec $C'_j = (y_{1j}, \ldots, y_{nj})$.

Le pivot de Gauss consiste à se ramener à un système triangulaire. On utilise la première ligne $L_1 : a_{11}x_{1j} + \cdots + a_{1n}x_{nj} = y_{1j}$ pour éliminer la dépendance en x_{1j} des autres lignes. Pour cela, il suffit de remplacer chaque ligne L_i par $L_i - \frac{a_{i1}}{a_{11}} L_1$. ($a_{11}$ est appelé pivot.) On utilise ensuite la deuxième ligne pour éliminer la dépendance en x_{2j} des lignes suivantes et ainsi de suite jusqu'à obtenir un système triangulaire qui devient facile à résoudre : la valeur x_{nj} est donnée par la dernière ligne. En reportant cette valeur dans l'avant dernière ligne, on obtient alors x_{n-1j}. En remontant ainsi de suite, on obtient $x_{n-2j}, \ldots, x_{1j}$.

Dans le cas où on tombe sur un pivot nul, il faut trouver une ligne qui fournit un pivot non nul et l'échanger avec la ligne courante. (On notera cependant, qu'une telle éventualité est peu probable.)

Pour obtenir le fichier entier, il faut effectuer les mêmes opérations sur les k systèmes linéaires $AB'_j = C'_j, 1 \leq j \leq k$.

3 Travail minimal

Le travail minimal consiste à programmer le codage et le décodage d'un fichier. Pour ces deux opérations, on utilisera un fichier de configuration `config.txt` contenant les nombres $p, q, k_m, g_1^d, \dots, g_{k_m}^d$. Le fichier sera dans un format textuel et contiendra un nombre par ligne sous forme décimale dans l'ordre $p, q, k_m, g_1^d, \dots, g_{k_m}^d$ (k_m désigne le nombre maximal de nombres dans un bloc).

L'opération de codage qui prend en argument un nom de fichier de configuration, un nombre de blocs n et un nom `f` de fichier à coder considérera `f` comme une suite de n blocs $B_1, \dots, B_n$ de $\lceil \frac{s}{n(l_q-1)} \rceil$ nombres de $l_q - 1$ bits où s désigne la taille du fichier en bits et l_q le nombre de bits de q. (Le dernier bloc sera éventuellement complété par des zéros.) Elle produira deux fichiers :

- un fichier texte `f.ava` contenant sous forme décimale avec un nombre par ligne les nombres $s, n, h(B_1), \dots, h(B_n)$,
- et un fichier binaire `f.dat` contenant sous forme binaire n combinaisons aléatoires à la suite. Chaque combinaison sera constituée des n coefficients $a_1, \dots, a_n$ de la combinaison suivis des k nombres $y_1, \dots y_k$ constituant la combinaison. Les a_i et les y_j sont des nombres de l_q bits. Chaque combinaison aura donc une taille de $(n+k)l_q$ bits exactement.

Le décodage consiste à reconstituer le fichier originel dans `f.dec` à partir du même fichier de configuration et des deux fichiers `f.ava` et `f.dat` codant un fichier. Un appel particulier permettra de plus de générer un fichier de configuration étant donné $l_p, l_q, kmax$. Fournir un programme permettant d'effectuer mot pour mot le test suivant sur un fichier `f` :

```
javac *.java
java Avalanche config 1024 256 3000 conf.txt
java Avalanche coder conf.txt 32 f
java Avalanche decoder conf.txt f
diff f f.dec
```

Le premier paramètre indique l'opération à effectuer. L'opération `config` indique de construire un fichier de configuration `conf.txt` avec $l_p = 1024, l_q = 256, k_m = 3000$. L'opération `coder` prend aussi en paramètre le nombre de blocs.

Il est important de respecter les formats de fichier spécifiés ci-dessus pour que votre programme puisse utiliser des fichiers de configuration, ou de codage qu'il n'a pas généré (comme les challenges proposés sur la page web de suivi).

L'opération de décodage produira une erreur si :

- une des combinaisons ne produit pas l'empreinte attendue par la fonction de hachage (on indiquera le numéro de la combinaison fautive),
- ou la matrice des coefficients des combinaisons linéaires n'est pas inversible.

Pour calculer sur des grands entiers, on utilisera la classe `BigInteger` de java ou la librairie GMP [4] en C. Pour trouver deux nombres p et q de l_p et l_q bits respectivement tels que $p = dq + 1$, on tire aléatoirement un nombre d pair de $l_p - l_q$ bits et un nombre premier q de l_q bits, et on construit $p = dq + 1$. Il suffit de quelques centaines d'essais pour tomber sur un nombre p premier. (Les deux librairies précédentes fournissent des fonctions de génération de nombres premiers aléatoires et de test de primalité ainsi que toutes les fonctions nécessaires au calcul modulo un grand entier comme l'élévation d'un nombre à une puissance modulo p ou le calcul d'un inverse modulo q).

Une des difficultés du sujet consiste à rendre les opérations de codage, de décodage et de vérification rapides. Une étude des temps de codage et de décodage d'un fichier pour diverses valeurs de n et k sera grandement appréciée. On donnera par exemple le débit moyen de décodage (nombre de kbits de fichier décodés par seconde) d'un fichier de plusieurs méga-octets pour $n = 40$. (On essaiera d'atteindre plus de 100 kbits/s.) Au besoin, on pourra augmenter l'espace mémoire utilisé par java avec l'option `-Xmx` de java (voir `man java`).

En particulier, on pourra utiliser l'optimisation suivante pour optimiser le temps de décodage (un temps de codage plus long est acceptable dans une application réelle). Pour n grand, le coût du pivot de Gauss devient prohibitif, on peut donc supposer k nettement plus grand que n. La vérification des combinaisons peut alors être optimisée en vérifiant plusieurs combinaisons d'un coup : il suffit pour cela de produire une combinaisons linéaire aléatoire de ces combinaisons et de la vérifier. Si cette combinaison de combinaisons est valide, elles le sont probablement toutes. Sinon, il faudra trouver la ou les combinaisons fautives. En supposant que peu de combinaisons sont corrompues, ce mécanisme peu accélérer grandement la vérification de l'intégrité des combinaisons.

Une autre difficulté réside dans la lecture et l'écriture au bit près dans un fichier. On fournira pour cela un module indépendant permettant de lire ou d'écrire dans un fichier une suite de nombres binaires de longueur en bits donnée. On utilisera de préférence une variable de type `int` pour stocker les bits intermédiaires qu'il faudra parfois stocker entre deux lectures ou deux écritures. Rappelons les opérations java pour manier les bits d'un `int` :

- La base hexadécimale s'utilise avec le préfixe `0x`, `0xf09` est par exemple l'entier positif de représentation binaire `111100001001`.
- `i & j` effectue le « et » bit à bit de `i` et `j`.
- `i | j` effectue le « ou » bit à bit de `i` et `j`.
- `i << j` décale les bits de `i` de `j` bits vers la gauche, en insérant à droite des `0`.
- `i >> j` décale les bits de `i` de `j` bits vers la droite, en insérant à gauche soit des `0` si $\texttt{i} \geq 0$, soit des `1` si $\texttt{i} < 0$.
- `i = b & 0xff` convertit un `byte` en `int` positif (les `byte` sont signés en java et la conversion directe en `int` recopie le signe).
- Un nombre négatif `i` est représenté en `int` par $2^{32}+\texttt{i}$, ainsi `-1 = 0xffffffff`.

4 Extension réseau

L'extension proposée ici consiste à programmer le client permettant de s'échanger des combinaisons linéaires avec d'autres clients en s'inspirant du protocole BitTorrent[1]. Chaque client sera capable de jouer le rôle de « tracker » (qui consiste à donner l'adresse d'autres clients auxquels se connecter), de source (tout client qui possède le fichier en entier devient une source) et de téléchargeur.

Pour télécharger un fichier, un client doit posséder le fichier `f.ava` et connaître l'adresse d'une source ou d'un client qui connaît une source. Il se connecte à ce client ou à cette source qui lui renvoie d'autres adresses de clients à qui se connecter. Chaque client se connecte ainsi à plusieurs autres clients.

Chaque client propose à tout client auquel il est connecté une combinaison aléatoire de toutes les combinaisons qu'il possède (les combinaisons d'une source sont les blocs eux-mêmes). Il calcule les coefficients exprimant cette combinaison comme combinaison des blocs originaux et l'envoie. Si le destinataire constate que cela n'augmente pas le rang de la matrice des combinaisons qu'il possède déjà, il refuse l'échange.

Un client ne sert des données qu'à quatre des clients auxquels il est connecté par périodes de dix secondes. Un client envoie des données en priorité aux trois qui lui ont le plus donné durant la période précédente. Toutes les dix secondes, ce choix est réévalué. De plus le client sert en permanence un quatrième pair sélectionné au hasard toutes les trente secondes.

Les connections entre clients s'effectueront au moyen de sockets TCP (voir la classe `java.net.Socket` de java) qui permettent d'échanger des données par le réseau de manière similaire à l'écriture et à la lecture dans un fichier. Pour accepter plusieurs connections, chaque client se comportera comme un serveur (voir la classe `java.net.Socket` de java).

Références

[1] B. Cohen. Incentives build robustness in BitTorrent. Workshop on Economics of Peer-to-Peer Systems, 2003. `http://www.bittorrent.com/`.

[2] C. Gkantsidis, P. Rodriguez. Network coding for large scale content distribution. IEEE Infocom, 2005. `http://research.microsoft.com/~pablo/avalanche.htm`.

[3] M. Krohn, M. Freedman, D. Mazieres. On-the-Fly Verification of Rateless Erasure Codes for Efficient Content Distribution. IEEE Symp. on Security and Privacy, 2004. `http://www.scs.cs.nyu.edu/~dm/papers/krohn:authcodes.pdf`

[4] Gnu Multiple Precision Arithmetic Library. `http://directory.fsf.org/gnump.html`.