Bitfountain

Amos Wenger & Rivo Vasta

—

Sommaire

I. Algorithme

1) Encodage Reed-Solomon

a) Implémentation choisie

L'implémentation choisie est inspirée du travail de James Planck (à travers ce document). La matrice génératrice est construite à partir de la matrice de Vandermonde (il s'agit de la même matrice que celle du cours), puis elle est réduite par l'algorithme de Gauss habituel. En effet, l'algorithme de Reed-Solomon fonctionne toujours pour toute matrice équivalente par lignes, et avoir une matrice réduite nous permet de gagner beaucoup de temps lors du décodage.

b) Paramètres du code

Nous avons choisi n = 223 mots de données par bloc, avec m = 32 mots de vérification par bloc, ce qui donne des blocs de k = 255 mots. On peut donc transmettre 223 bytes de données en envoyant un bloc de 255 bytes, en tolérant jusqu'à 31 bytes de perte (c'est à dire 12.15% en théorie). Ce code a un rapport de $223/255 \approx 0.87$

c) Entrelacement

L'algorithme de Reed-Solomon permet de récupérer des pertes à l'intérieur des blocs, mais pas de reconstituer les données si des blocs entiers sont perdus. On doit donc entrelacer les données afin d'envoyer des portions de blocs dans chaque paquet. Ainsi, si des paquets sont perdus, cela correspondra à quelques bytes manquants dans les blocs.

La taille d'envoi des paquets étant limitée à 1400 bytes par l'énoncé, et comme nous avons un header de 13 bytes (1 byte pour le type de packet, 8 bytes pour l'identifiant théoriquement unique du fichier, et 4 bytes pour le numéro du paquet), on envoie des données de 1387 blocs différents dans un seul paquet.

Pour décoder il faut évidemment attendre d'avoir atteint le paquet 255 pour commencer à décoder quoi que ce soit. On détecte les pertes grâce aux discontinuités qui prennent place dans les numéro de paquets lorsque l'un d'entre eux n'est pas transmis. Si on reçoit le paquet 69 et que le dernier paquet reçu était le 67, on suppose qu'on a perdu le 68. (Note : cela ne marche bien évidemment que si la réception des paquets est ordonnée, ce qui est le cas sur TCP/IP).

2) Décodage Reed-Solomon

a) Le principe général

Le concept général de Reed-Solomon est d'envoyer (ou de stocker) un système d'équations qui est sur-spécifié (certaines informations sont redondantes) de manière particulière, afin que si un nombre raisonnable d'informations sont perdues, ont arrive quand même à résoudre l'équation avec les informations qui sont passées.

Une approche naïve est, pour chaque bloc qu'on veut décoder, de créer une matrice à partir de la matrice génératrice, à laquelle on enlève les lignes correspondants aux paquets perdus pendant la transmission, augmentée d'une colonne contenant les données reçues, puis d'appliquer l'algorithme de réduction de Gauss sur cette matrice. Le résultat est, à gauche, la matrice identité, et dans la colonne de droite, les données décodées.

b) Algorithme de Gauss optimisé pour matrices quasi-identité

L'algorithme implémenté possède la propriété de construire la matrice génératrice comme une matrice 255×223, dont la partie haut de 223×223 est une matrice identité

Lors de la perte de paquets (dans la réalité ou par simulation), le récepteur établit la liste des paquets manquants (ceux-ci sont en effet numérotés), et met à 0 les lignes correspondantes dans la matrice génératrice. La matrice se présente alors de la façon suivante : sur les 223 premières lignes, on sait que la ligne est soit une ligne de la matrice identité (c'est-à-dire que le terme de la diagonale est le seul non nul et égal à 1), soit nulle (0 partout). Sur les lignes suivantes, la lignes est soit une ligne comportant des informations chiffrées, soit une ligne nulle.

L'algorithme optimisé procède alors de la façon suivante : pour les 223 premières lignes, il se contente de regarder la valeur de la cellule de la diagonale. Si celle-ci est nulle, il faut l'échanger avec la première ligne non nulle et non identité de la matrice (donc à partir de la 224ème ligne), à condition toutefois que celle-ci soit « éligible ». Une ligne « éligible » est une ligne dont le terme se trouvant sur la colonne du terme de la diagonale de la ligne qu'on souhaite échanger est non nul après sa réduction à droite par les lignes du dessus. En effet, si celui-ci venait à être nul après réduction, alors cela signifierait qu'on échangerait une ligne nulle avec une autre ligne nulle (après réduction), ce qui rend la résolution du système d'équations, et donc le décodage, impossible par l'algorithme de Gauss (le programme renvoie alors une erreur de « matrice singulière »).

Dans la pratique, lors de la rencontre d'une ligne nulle, on prend la première ligne de rang strictement supérieur à 223 qui n'ait pas déjà été l'objet d'un échange avec une autre ligne (sinon c'est qu'elle est nulle ; c'est notre tableau d'entiers `takenCandidates` qui retient les lignes déjà prises), et on en fait une copie provisoire sur laquelle faire des tests d'éligibilité sans toucher à la vraie matrice. On simule une réduction sur cette ligne (comme si l'échange était effectif et qu'on en était à une étape de réduction), et si après la simulation le terme qui se trouve en position d'être sur la diagonale en cas d'échange de ligne n'est pas nul, alors la ligne est acceptée et l'échange de lignes se fait réellement. Sinon, on continue avec les lignes suivantes. Si aucune ligne candidate n'a été trouvée, on part du postulat que cette situation témoigne d'un nombre de lignes nulles (donc de paquets perdus) strictement supérieur à ce que l'algorithme peut détecter, c'est-à-dire strictement supérieur au nombre de colonnes, c'est-à-dire strictement supérieur à 223.

c) Décodage parallèle

La méthode de décodage naïve présenté en a) est assez lente. Heureusement, on peut tirer profit de l'entrelacement que l'on utilise pour transmettre les blocs en plusieurs paquets. Puisqu'on envoie 1387 blocs à la fois, cela signifie également que ces 1387 blocs perdent des données aux mêmes lignes, correspondant aux paquets qui n'ont pas été transmis. Ainsi, il est possible de créer une matrice augmentée des 1387 colonnes correspondant aux blocs reçus et de décoder tous ces blocs en un passage de Gauss amélioré (décrit ci-dessus). En test, cette optimisation nous a permis de dépasser les 12 Mio/s en taux de transfert local + décodage.

II. Architecture réseau

1) Le protocole

Le protocole se base sur la transmission de paquets (dont une architecture est décrite par la super-classe `Packet`. Chaque paquet contient un identifiant unique, son type (`PacketType`), qui décrit en un octet l'information qu'il porte. Chacun des paquets comporte également une méthode `write()`, qui a pour fonction d'envoyer, via un flux `OutputStream`, les données du paquet sur le réseau. Envoyer un paquet du client au serveur ou inversement revient dans la pratique à créer une nouvelle instance du paquet qu'on souhaite envoyer et d'appeler dessus la méthode membre `write()` avec pour paramètre le flux de sortie (`outputStream`, un flux par client).

Afin de distinguer le protocole des autres qui pourraient entrer en conflit, chaque client qui tente de se connecter au serveur en l'utilisant doit envoyer un nombre particulier de reconnaissance qui atteste qu'il souhaite bien se connecter à notre serveur. Il doit donc s'identifier par un nombre unique dont l'écriture hexadécimale est 0xDEADBABE et qui lui assureront l'acceptation par le serveur.

a) Identifier un fichier

Le paquet `FileHeader` se charge de donner une description du fichier. Il est constitué du nom du fichier, de sa taille en octets, et d'un numéro d'identification théoriquement unique. Lorsqu'il transmet des informations sur le réseau, il transmet d'abord la taille du fichier, puis son nom, ceci afin d'être sûr que l'information sur la taille arrive même si le nom dépasse la capacité du bloc d'information qui lui est réservé.

b) Transmission du catalogue

Lorsque des fichiers sont disponibles pour les utilisateurs (car l'énoncé spécifie qu'un fichier mis en ligne est disponible pour tous les utilisateurs connectés à ce moment-là), le serveur envoie un paquet de type `FileList`, qui est une liste de `FileHeader`, permettant ainsi au client de connaître la liste des fichiers disponibles.

c) Demander un fichier

Le client qui souhaite télécharger un des fichiers à sa disposition envoie, par l'intermédiaire de l'interface, un paquet `FileRequest` au serveur. Celui-ci contient l'identification théoriquement unique qui permet au serveur d'envoyer le bon fichier.

d) Envoi progressif d'un fichier

Lorsqu'un fichier est envoyé à un utilisateur, celui-ci est décodé en petites parties décrites par les paquets `FilePart`, qui contiennent l'identifiant théoriquement unique, le numéro du paquet, et la partie du fichier sous forme d'un bloc d'octets.

L'envoi de chaque paquet se fait par la méthode `step()` (« étape »), qui est appelée par chaque tâche. Ainsi, plusieurs envois peuvent s'effectuer, pas en même temps, mais successivement par petits avancements les uns après les autres.

e) Fichier introuvable

Dans le cas où le client demanderait un fichier au serveur que celui-ci ne possèderait plus, un paquet `FileNotFound` est renvoyé pour lui signifier cet état de fait, avec l'identification du fichier manquant. L'application client peut alors comparer l'identification avec celle des fichiers de la liste des fichiers disponibles pour retrouver quel fichier a été demandé au serveur alors que celui-ci ne le possédait plus.

Ce cas de figure est rare, il n'arrive que lorsque le client possède une liste de fichiers disponibles qui n'est plus en accord avec la vraie disponibilité des fichiers sur le serveur. Cette liste du côté client étant mise à jour à chaque nouveau fichier que le serveur reçoit, ceci ne peut arriver que si aucun fichier n'est téléversé sur le serveur durant une période significative (donc aucune mise à jour chez le client), et que pour une raison ou une autre, certains fichiers disponibles sur le serveur soient enlevés de celui-ci (une telle action n'ayant pas été implémentée, le cas de figure ne devrait en fait jamais arriver).

f) Annulation d'un transfert

Un utilisateur peut vouloir annuler le transfert d'un fichier, qu'il s'agisse d'une mise à disposition sur le serveur ou d'un rapatriement (téléchargement). Dans le cas où il s'agit d'une mise en ligne, l'action d'annulation enlève tout simplement la tâche en cours d'exécution (en train d'envoyer le fichier) de la liste des tâches du client (qui sont distinguables à travers l'identificateur théoriquement unique du fichier).

Dans le cas où il s'agit d'un téléchargement, un paquet `CancelTransfer` est envoyé au serveur avec l'identifiant théoriquement unique du fichier en cours de transfert. La manipulation est alors la même du côté serveur : la liste des tâches en cours est parcourue et la tâche qui est associée au même identifiant que le fichier se voit retirée de la liste.

2) Le serveur

Le serveur fonctionne selon deux fils d'exécution (*threads*), détaillés ci-après selon leur fonction.

a) Accepter des clients

Le premier fil d'exécution est décrit dans la classe `ClientListener`. Il est à l'écoute des nouveaux utilisateurs qui souhaitent se connecter au serveur. Lorsqu'un client tente de s'y connecter, il vérifie que le numéro d'identification est le bon (auquel cas le client est rejeté), puis le rajoute à la liste des clients (voir le paragraphe d)) et affiche un message d'information dans la console du serveur (voir le paragraphe Console).

b) Traiter les demandes des clients

Le second fil d'exécution se trouve dans la classe ClientPoller, qui est à l'écoute des demandes des clients à travers les paquets de requête qu'ils envoient par les flux. Elle analyse le premier octet de chaque paquet reçu pour en déterminer le type, grâce auquel elle construit un paquet dudit type qu'elle envoie à la méthode `handlePacket()`, qui se charge de prendre les décisions adéquates selon le type de paquet.

c) Tâches parallèles

Chaque envoi au client est géré par une classe qui hérite de `Task` et qui implémente une fonction `step()`. La méthode `step()` contient une série de petits travaux à exécuter. Le serveur possède une liste de tâches (*tasks*) à exécuter, dont il exécute à la suite les méthodes `step()`, permettant ainsi d'exécuter un peu de chaque tâche à chaque tour.

d) Simulation de l'état du réseau

Le serveur possède une classe `ClientInfo` qui contient toutes les informations dont il dispose sur un client particulier : son nom, son *socket* (interface logicielle de communication), ses flux d'entrée et de sortie, la liste des fichiers qui lui sont téléchargeables et enfin son *buffer*, liste de morceaux de fichiers.

Pour simuler le réseau, le serveur dispose d'une liste des clients qui s'y sont connectés. Dans l'esprit de l'énoncé, il est considéré qu'un client figurant sur cette liste est un client connecté. Un

client que se déconnecte doit en disparaître (voir paragraphe Déconnexion ci-dessous). On part du principe que les pertes ne peuvent intervenir que sur les fichiers en cours de téléchargement (du serveur vers le client), et qu'elles ne touchent pas les paquets de transmission d'information (telles que les envois de la liste des fichiers disponibles pour l'utilisateur).

3) Le client

a) Connexion et identification

L'inscription d'un client sur le serveur se fait par le biais de la méthode `connectToServer()` de la classe `ClientCommunication` (classe qui gère toutes les interactions entre un utilisateur et le serveur). Pour se connecter, l'utilisateur doit entrer un nom d'utilisateur, une adresse et un port. Pour lui faciliter la tâche, les champs sont préremplis par des valeurs par défaut fonctionnelles. Le programme du côté client se charge également d'envoyer dans le flux de sortie le nombre caractéristique qui lui permettra de se connecter au serveur.

b) Le catalogue

Le catalogue est propre à chaque utilisateur. Lorsqu'un utilisateur envoie un fichier, il arrive dans l'ensemble des catalogues des utilisateurs qui sont connectés. Ainsi, si un utilisateur n'était pas connecté à ce moment-là, il ne verra pas ce fichier disponible dans son catalogue.

c) Tâches parallèles

Tout comme le serveur, le client possède une liste de tâches qui sont autant de fichiers qu'il est en train d'envoyer au serveur. Là aussi, une boucle exécute la méthode de `step()` de chaque `Task`, afin de faire avancer un petit peu chaque tâche à la suite.

d) Déconnexion

Dans la réalité, une déconnexion n'est pas toujours prévisible et peut être brutale. Le serveur ne doit ainsi pas compter sur la réception d'un paquet de départ de la part d'un client qui se déconnecte.

Aussi, pour savoir qu'un client s'est déconnecté, le serveur ne compte-t-il que sur le fait qu'une exception aura été levée, soit une `SocketException`, soit une `IOException`, qu'il peut attraper pour enlever le client de la liste des clients.

III. Interface

1) Client

L'interface client se présente comme une fenêtre constituée principalement de trois onglets décrits individuellement dans les trois points qui suivent.

a) Téléchargements

L'onglet Téléchargements présente une liste des fichiers en cours de téléchargement. Pour chaque fichier, l'utilisateur dispose d'informations complètes sur son nom, sa taille et sa vitesse de téléchargement, d'une barre de progression pour mieux visualiser l'avancement, et de trois boutons. Pendant le téléchargement, il peut annuler le transfert (voir à ce sujet le paragraphe correspondant). Après celui-ci, il peut l'ouvrir directement depuis l'interface, et/ou le supprimer de la liste des fichiers, le transfert étant terminé.

b) Catalogue

L'onglet Catalogue présente la liste des fichiers disponibles au téléchargement sur le serveur pour le client. Il peut en sélectionner un et dispose alors d'un bouton qui lui permet de lancer le téléchargement s'il le désire.

c) Téléversements

L'onglet Téléversements permet au client d'héberger ses fichiers sur le serveur. Un bouton « Téléverser » lui ouvre une boîte de dialogue d'ouverture de fichiers d'où il peut sélectionner un ou plusieurs fichiers à envoyer sur le serveur. Une fois les fichiers choisis et validés, ceux-ci apparaissent dans la liste des fichiers en cours d'hébergement, depuis laquelle l'utilisateur peut annuler le transfert s'il le souhaite.

2) Serveur

a) Console

Il s'agit d'une console graphique qui affiche des informations à propos des différents évènements qui se passent sur le serveur, tels que la connexion d'un nouveau client, l'envoi ou la réception d'un fichier par celui-ci, etc.

b) Paramètres

Un onglet de l'interface du serveur permet à l'administrateur d'icelui de régler le pourcentage de perte afin de simuler un réseau sans fil réel avec paquets égarés.

IV. Choix de conception

1) Conventions d'écriture

Nous avons suivi les suggestions quelque peu allégées de l'outil CheckStyle, afin de garantir la lisibilité et l'uniformité du code, ainsi que sa couverture en documentation.

2) Licence

L'ensemble du programme est disponible sous licence WTFPL, dont on pourra trouver les différentes conditions à cette page.