Université de Strasbourg
UFR Mathématique et Informatique

Licence 3 Informatique
Année 2011/2012

Réseaux et Protocoles

TP03 : Les codes correcteurs d'erreur

1 Introduction

Ce TP consite à effectuer un transfert de fichiers par l'intermédiaire de sockets UNIX en mode datagramme à travers un support non fiable. Chaque exercice présenté dans la suite vous propose d'implémenter un code correcteur d'erreur différent pour détecter (voir corriger) les erreurs dues au support de transmission non fiable. Pour chaque exercice, vous allez utiliser trois programmes différents : l'émetteur, le récepteur et le médium non fiable. Ce dernier sera un processus intermédiaire entre un émetteur et un récepteur qui introduira des erreurs dans le flot de données. Il y aura éventuellement plusieurs types d'erreur possibles : erreurs simple (1 bit par mot de code) ou en rafale (avec des fréquences plus ou moins grande).

Plus explicitement, nous supposons que nous devons transférer le fichier *FileToSend*. Pour simplifier, le transfert aura lieu à l'intérieur d'un répertoire : les fichiers de départs (*FileToSend*) et d'arrivée (*FileReceived*) seront dans le même répertoire. Le transfert se décompose comme illustré sur la figure 1 :

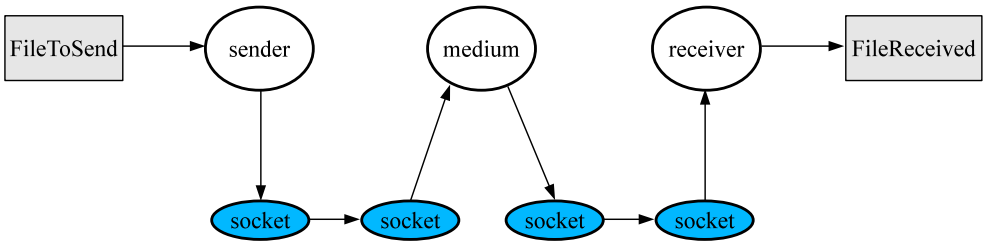

FIGURE 1 –

Pour réaliser les exercices présentés dans la suite, on vous propose d'utiliser un code source qui implémente déjà tout ce qui précède (programmes émetteur, médium et récepteur avec leurs interactions). Ce code est disponible à l'adresse suivante :

http://www.droth.eu/docs/rp/CodeCorrecteur.tgz

Si vous ne savez pas ouvrir une archive TAR compressée avec GNU Zip (extension .tgz ou .tar.gz), lisez la documentation sur la commande `tar` dans le manuel (`man tar`).

1.1 Description des trois programmes

Le répertoire *CodeCorrecteur* contient les programmes *sender*, *receiver* et *medium*.

1.1.1 Sender

Le programme *sender* est décrit dans les sources : *sending.c* (module principal), *coding.c*, *logic.c* et *sockettoolbox.c*. Pour lancer *sender* : `./sender FileToSend`

L'émetteur lit le fichier *FileToSend* par blocs de *MAX_MESSAGE* octets (50 par défaut). Ces blocs représentent l'information utile (i.e. les bits de donnée). Pour aller au plus simple, on considérera que, pour la transmission, les données sont groupés en mot de 8 bits. Evidemment, avant de transmettre, il faut coder le message, i.e. ajouter des bits de contrôle. Chaque mot de données (8 bits) est encapsulé dans une structure plus grande capable de contenir les bits de donnée et les bits de contrôle. Dans le cas de ce programme, nous avons choisi d'encapsuler chaque caractère (8 bits) dans un short (16 bits). Pour l'occasion l'entier court (short) a été rebaptisé *CodeWord_t* voulant dire *mot du code*. Au codage, les 8 bits de données sont encapsulés dans le *CodeWord_t* par la fonction *copyDataBitsCoding* (du fichier *coding.c*). Le schéma ci-dessous illustre ce processus :

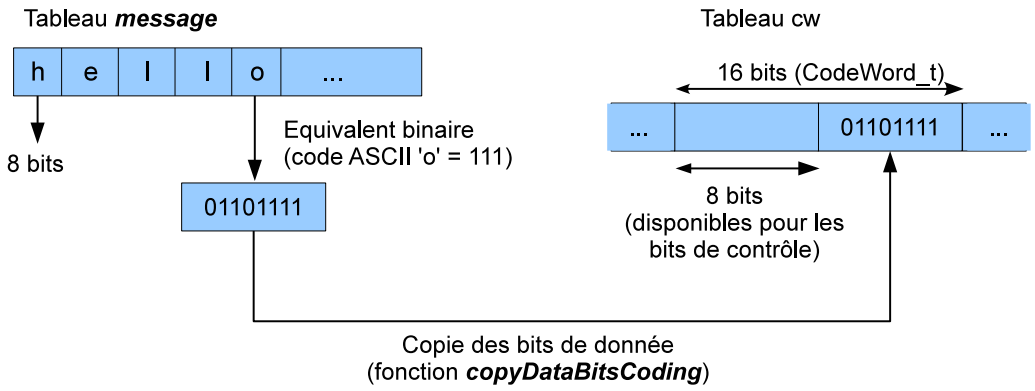

FIGURE 2 –

C'est la raison pour laquelle chaque bloc de *MAX_MESSAGE* octets lus dans *FileToSend* fait l'objet d'une transmission d'un bloc de *MAX_MSG_CODE* octets (100 par défaut).

1.1.2 Medium

Le programme *medium* est décrit dans les sources : *medium.c* (module principal), *error.c*, *logic.c* et *sockettoolbox.c*. Pour lancer medium :

- *medium 0* : n'introduit aucune erreur.
- *medium 1* : introduit une erreur d'un bit dans chaque mot du code (chaque CodeWord_t, i.e. chaque short).
- *medium 2* : Pour *ERROR_RATE=0* n'introduit aucune erreur et pour *ERROR_RATE=1* introduit une erreur dans chaque caractère (une erreur par bloc de 8 bits). Toutes les situations intermédiaires sont possibles.
- *medium 3* : introduit un caractère d'erreur dans une chaîne.
- *medium 4* : Pour *ERROR_RATE=0* n'introduit aucune erreur et pour *ERROR_RATE=1* change tous les caractères. Ici aussi, toutes les situations intermédiaires sont possibles.

1.1.3 Receiver

Le programme *receiver* est décrit dans les sources : *receiving.c* (module principal), *decoding.c*, *logic.c* et *sockettoolbox.c*. Pour lancer *receiver* :

```
./recepteur FileReceived
```

Cette commande enregistre dans le fichier *FileReceived* le contenu décodé de ce qui est reçu sur la socket.

1.1.4 Exemple

Par exemple pour envoyer le fichier *LISEZ-MOI* sur un médium sans erreurs et l'enregistrer sous le nom de *totorec* faites :

```
./medium 0 &
./receiver totorec &
./sender LISEZ-MOI
```

1.2 Ce qui vous aurez à faire

Dans chacun des exercices suivant vous devrez implémenter le code correcteur proposé. Pour le codage, vous devrez écrire la fonction *computeCtrlBits* présente dans le fichier *coding.c*. Pour le décodage, les bits de contrôle doivent être extraits et interprétés par les fonctions *errorCorrection* (pour l'exercice 3) et *thereIsError* (pour les exercices 1 et 2) du fichier *decoding.c*.

Pour l'instant ces fonctions ne font rien (elles sont d'ailleurs commentées), c'est à vous de les remplir en fonction du codage demandé. Pour la syntaxe, vous pouvez vous inspirer des fonctions *copyDataBitsCoding* et *copyDataBitsDecoding*.

2 Exercices

Exercice I : bit de parité

Il s'agit de détecter les erreurs introduites par le canal grâce à l'ajout d'un bit de parité à chaque caractère au niveau de l'émetteur. Les mots du code ont donc une longueur de 9 bits : 8 bits d'information + 1 bit de contrôle. Comme mentionné précédemment, chaque blocs de 8 bits d'information est automatiquement placé dans un bloc de 16 bits (cela est réalisé par la fonction *copyDataBitscoding*). Pour un bloc de 8 bits d'information, on dispose donc de 8 bits supplémentaires pour stocker les bits de contrôle (cf. figure 2). Pour simplifier, seul le premier bit de l'espace réservé au bits de contrôle sera utilisé (comme bit de parité). De son côté, le récepteur devra afficher les caractères qu'il détecte comme erronés.

Il faudra afficher toutes les chaînes dans lesquelles le médium aura introduit des erreurs, ainsi que toutes les chaînes dans lesquelles le récepteur aura détecté des erreurs. Les deux listes devraient être identiques. Quels sont les types d'erreurs que l'on peut détecter avec ce codage ?

Exercice II : codage polynomial

Il s'agit de détecter les erreurs introduites par le canal grâce à un codage polynomial. Pour ce faire, vous pouvez utiliser le polynôme générateur suivant :

$$g(x) = x^8 + x^4 + x^3 + x^2 + 1$$

Les mots codés auront une longueur de 16 bits (8 bits d'information + 8 bits de contrôle). De même que pour l'exercice 1, l'émetteur doit coder les données à l'émission et le récepteur doit décoder les données à la réception et vérifier si les données reçues sont ou non erronées.

Il faut détecter toutes les erreurs introduites par le médium dans les données. Il faudra afficher toutes les chaînes dans lesquelles le médium aura introduit des erreurs, ainsi que toutes les chaînes dans lesquelles le récepteur aura détecté des erreurs. Les deux listes devraient être identiques. Quels sont les types d'erreurs que l'on peut détecter ?

Exercice III : codage de Hamming

Il s'agit maintenant de corriger les erreurs introduites par le canal grâce à un codage de *Hamming*.

On considère C, un code (n, k) systématique, de matrice génératrice $G = (I_k, A^t)$, de poids minimum p.

1. Quelle est la valeur du kième bit d'une combinaison linéaire quelconque des k-1 premières lignes de G ?
2. On construit la matrice génératrice $G' = (I_{k-1}, A'^t)$ où A'^t est obtenue en supprimant la dernière ligne de A^t. Quel est le poids minimum de ce code ? Le code C' engendré par G' est appelé un code raccourci de C.
3. Appliquer cette construction au code de Hamming (7,4) : donnez G' et H'. Combien d'erreurs peut-il corriger ?
4. Si on applique 3 fois cette opération de raccourci à partir de ce code de Hamming, quel code obtient-on ? Quel est son poids minimum et ses capacités de détection et de correction ?
5. Y a-t-il plusieurs matrices G possibles pour la question 3) ? Si oui peut-on obtenir des réponses différentes à la question 4) ? Sinon justifiez.
6. Pour quelles valeurs de n existe-t-il un code de Hamming (n,k) ?
7. Donnez une méthode pour construire quelque soit k un code linéaire (n, k) capable de corriger une erreur. Quelle est la longueur du code obtenu si on veut protéger k bits d'information, $k = 8$?
8. Ecrire la matrice génératrice et de parité du code obtenu à la question précédente.
9. Utiliser ce code pour protéger les données envoyées sur le médium. En particulier, l'émetteur doit insérer des bits de contrôle dans les données transférées et le récepteur doit décoder les données à la réception, c'est-à-dire interpréter les bits de contrôle, extraire l'information utile et la corriger. Il faut parvenir à ce que *FileToSend* et *FileReceived* soient exactement identiques, malgré les erreurs de transmission.

3 Annexes

Le codage et le décodage impliquent nécessairement la manipulation de nombres bit par bit. Pour cela, on dispose de quelques outils.

Décalage bit à bit

On rappelle que pour décaller tous les bits d'un nombre *n* de *p* bits vers le côté des bits de poids fort, on utilise l'opérateur «. Pour le décalage vers les bits de poids faible on utilise l'opérateur ».
Par exemple si n = 46 est un nombre de type char, alors on peut le représenter en binaire sous la forme 00101110.

```
n    : 00101110
n>>1 : 00010111
n<<1 : 01011100
```

En particulier, 1«n représente le nombre 2^n et n»1 représente le nombre $\frac{n}{2}$.

Opérateurs binaires

Les opérateurs de bits exécutent les opérations logiques ET, OU, OU exclusif (XOR) et NON sur tous les bits, pris un à un, de leurs opérandes entières. Le tableau ci-dessous rappelle la syntaxe et le comportement des différents opérateurs binaires.

&	ET logique	Retourne 1 si les deux bits de même poids sont à 1
|	OU logique	Retourne 1 si l'un ou l'autre des deux bits de même poids est à 1 (ou les deux)
^	OU exclusif	Retourne 1 si l'un des deux bits de même poids est à 1 (mais pas les deux)
~	NON logique	Inverse tous les bits

Le module logic.c

Le module *logic.c* contient quelques fonctions permettant par exemple d'afficher les bits d'un caractère ou d'un entier court (la fonction *CheckBitABit)*, de rendre la valeur du n-ième bit d'un nombre (la fonction *getNthBit*) ou alors de mettre la valeur de ce n-ième bit à 0 ou 1 (la fonction *setNthBitW* ou *setNthBitCW*). Et d'autres fonctions encore ...