Travaux Pratiques Réseaux

Les codes détecteurs et correcteurs d'erreur

1 Ce que doivent faire les programmes

1.1 La partie commune aux deux manips

Ce TP sera constitué de deux manips. Dans les deux manips, l'opération à effectuer (le prétexte du TP) sera un transfert de fichiers par l'intermédiaire de sockets (Unix) entre un processus émetteur et un processus récepteur. Ce transfert se fera par l'intermédiaire d'un support non-fiable.

Ce support non-fiable sera modélisé par un troisième processus intermédiaire entre émetteur et récepteur et qui introduira des erreurs dans le flot de données. Il y aura éventuellement plusieurs types d'erreurs possibles : erreurs simples (1 bit par mot de code) ou en rafale (avec des fréquences plus ou moins grandes).

Plus explicitement, nous supposons que nous devons transférer le fichier `FichEmis`. Pour simplifier, le transfert aura lieu à l'intérieur d'un répertoire : les fichiers de départ (`FichEmis`) et d'arrivée (`FichRecu`) seront dans le même répertoire. Le transfert se décompose comme suit (cf. figure 1) :

1. Le processus `emetteur` crée et attache une socket Unix `sock_em` de type `SOCK_DGRAM` dans le répertoire où il est lancé. Il ouvre le fichier `FichEmis` en lecture, lit les données, les code, et transfère les données codées vers la socket Unix `s_medem` ouverte par le processus `medium`.
2. Le processus `recepteur` crée et attache une socket Unix `sock_rec` de type `SOCK_DGRAM` dans le répertoire où il est lancé (le même que celui de l'émetteur, pour simplifier). Il ouvre le fichier `FichRecu` en écriture, attend les données venant de la socket `s_medrec` ouverte par le processus `medium`. Il décode les données, détecte ou corrige les erreurs, extrait les données et les écrit dans le fichier `FichRecu`.
3. Le processus `medium` crée et attache deux sockets Unix `s_medrec` et `s_medem` de type `SOCK_DGRAM` dans le répertoire où il est lancé (toujours le même). Ensuite les données de la socket `s_medem` sont écrits dans `s_medrec`. Entre les deux sockets, l'information aura subi des changements.

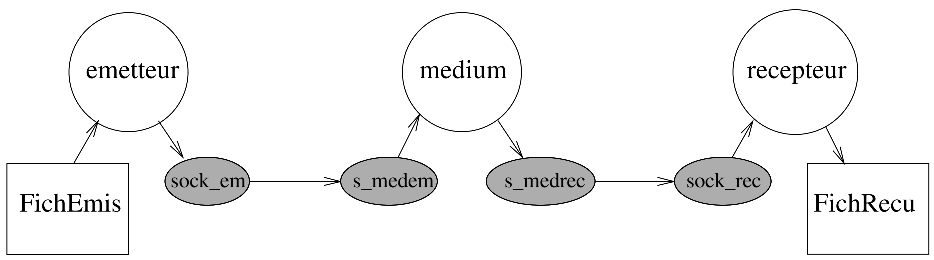

FIG. 1: Les cercles représentent des processus, les rectangles des fichiers et les ovales des sockets. Un processus `emetteur` envoie un fichier vers le processus `recepteur` par l'intermédiaire d'un processus `medium` qui modifie plus ou moins le flot de données.

1.2 Manip 1

1.2.1 Objectif

Il s'agit de *corriger* les erreurs introduites par le canal grâce à un codage de *Hamming*. Le processus `emetteur` doit insérer des bits de contrôle dans les données transférées et le processus `recepteur` doit décoder les données à la réception, c'est-à-dire interpréter les bits de contrôle, extraire l'information utile et la corriger.

1.2.2 Résultats attendus

Il faut parvenir à ce que `FichRecu` et `FichEmis` soient exactement identiques malgré les erreurs de transmission. Quels sont les types d'erreur que l'on peut corriger ?

1.3 Manip 2

1.3.1 Objectif

Il s'agit de *détecter* les erreurs introduites par le canal grâce à un codage polynomial. De même que pour la manip 1, le processus `emetteur` doit coder les données à l'émission et le processus `recepteur` doit décoder les données à la réception et vérifier si les données re çues sont ou non erronées.

1.3.2 Résultats attendus

Il faut détecter toutes les erreurs introduites par le medium dans les données.

On pourra, par exemple, afficher toutes les chaînes dans lesquelles le medium aura introduit des erreurs, ainsi que toutes les chaînes dans lesquelles le récepteur aura détecté des erreurs. Les deux listes devraient être identiques.

On pourra aussi comparer les chaînes dans lesquelles le récepteur aura détecté des erreurs et le résultat de la commande

```
diff FichEmis FichRecu
```

Quelle est la différence entre les deux démarches ?

Quels sont les types d'erreur que l'on peut détecter ?

2 Ce qui a déjà été fait

Pour réaliser les manips ci-dessus, vous avez le choix :

- Vous pouvez tout programmer à partir de rien, ou alors à partir de la manip 15 du TP sockets. Dans ce cas, le reste de ce document ne vous concerne pas.
- Vous pouvez aussi commencer à partir d'un programme tout prêt où tout ce qui précède a déjà été réalisé, à l'exception du calcul des bits de contrôle (du côté émission), de l'interprétation de ces bits, de la correction et de la détection proprement dites (du côté de la réception).

  Tout se trouve dans le répertoire :

  ```
  ~habibi/public/CodeCorrecteur.d
  ```

  Le reste de ce document décrit ce qui a déjà été réalisé dans ces programmes et ce qu'il vous reste encore à faire.

2.1 Description des trois programmes

Le répertoire `CodeCorrecteur.d` contient les programmes `emetteur`, `recepteur` et `medium`.

2.1.1 `medium`

Le programme `medium` est décrit dans les sources : `Medium.c` (module principal), `Erreur.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 de code (chaque CarCode_t, i.e. chaque short)

– `medium 2` : pour `TAUX_ERREUR=0` n'introduit aucune erreur et pour `TAUX_ERREUR=1` introduit une erreur dans chaque caractère. Toutes les situations intermédiaires sont possibles.

– `medium 3` : introduit un caractère d'erreur dans une chaîne.

– `medium 4` : pour `TAUX_ERREUR=0` n'introduit aucune erreur et pour `TAUX_ERREUR=1` change tous les caractères. Là aussi, toutes les situations intermédiaires sont possibles.

`TAUX_ERREUR` est définie dans le fichier `Erreur.h`.

2.1.2 `recepteur`

Le programme `recepteur` est décrit dans les sources : `Reception.c` (module principal), `Decodage.c`, `Logic.c` et `SocketToolbox.c`

Pour lancer `recepteur` :

```
recepteur FichRecu
```

Cette commande enregistre dans le fichier `FichRecu` le contenu décodé de ce qui est reçu sur la socket `sock_rec`.

2.1.3 `emetteur`

Le programme `emetteur` est décrit dans les sources : `Emission.c` (module principal), `Codage.c`, `Logic.c` et `SocketToolbox.c`.

Pour lancer `emetteur` :

```
emetteur FichAEmettre
```

Cette commande lit le fichier `FichAEmettre` code les données et les expédie sur la socket `sock_em`.

2.1.4 Exemple

Par exemple pour envoyer le fichier `LISEZ-MOI` et l'enregistrer sous le nom `totorec` faites :

1. `medium &`
2. `recepteur totorec &`
3. `emetteur LISEZ_MOI`

2.2 Pour traiter les bits individuellement

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

2.2.1 Décalage bit à bit

On rappelle que pour décaler 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 .

$$\begin{aligned} n &: 00101110 \\ n >> 1 &: 00010111 \\ n << 1 &: 01011100 \end{aligned}$$

En particulier `1«n` représente le nombre 2^n, `m«n` représente le nombre $m \cdot 2^n$ et enfin `m»n` représente le nombre $m/2^n$.

2.2.2 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 `NiemeBit`) ou alors de mettre la valeur de ce n-ième bit à 0 ou à 1 (la fonction `SetNiemeBitC` ou `SetNiemeBitCC`). Et d'autres fonctions encore ...

2.3 Codage-Décodage

L'émetteur lit le fichier `FichEmis` par blocs de `MAX_MESSAGE` octets (Ici `MAX_MESSAGE` vaut 50). Ces blocs représentent l'information utile.

Pour aller au plus simple, on considérera que, pour la transmission, les données sont groupés en mots de 8 bits.

Évidemment, avant de transmettre, nous devons 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 données 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 a été rebaptisé `CarCode_t` voulant dire "caractère codé".

Au codage, les 8 bits de données sont encapsulés dans le `CarCode_t` par la fonction `RecopieBitsDonneesEnc` (du fichier `Codage.c`). Au décodage, les 8 bits de données sont extraits du `CarCode_t` par la fonction `RecopieBitsDonneesDec` (du fichier `Decodage.c`).

C'est la raison pour laquelle chaque bloc de `MAX_MESSAGE` octets lus dans `FichEmis` fait l'objet d'une transmission d'un bloc de `MAX_MSG_CODE` octets. (Ici `MAX_MSG_CODE` vaut 100)

3 Ce qui reste à faire

Les bits de contrôle doivent être calculés par la fonction `CalculBitsCtrl` dans le fichier `Codage.c`.

Pour le décodage, les bits de contrôle doivent être extraits et interprétés par les fonctions `CorrigeErreurs` (pour la manip1) et `IlyaErreur` (pour la manip 2) dans le fichier `Decodage.c`

Pour l'instant ces fonctions ne font rien. A vous de les remplir. Pour la syntaxe, vous pouvez vous inspirer des fonctions `RecopieBitsDonneesEnc` et `RecopieBitsDonneesDec`.

4 Évaluation

Ces manips ne seront pas sanctionnées par une note. Mais les programmes écrits pendant ce TP seront probablement utilisés dans un autre TP (HDLC) qui, lui, sera effectivement noté.