Codes Correcteurs

Julie Badets, Corentin Frade, Quentin Rouland, Émeric Tosi

1er avril 2014

Sommaire

Introduction

Un code correcteur est utilisé pour transmettre un message dans un canal bruité. Il permet de reconstituer le message émis même si des erreurs (en nombre limité), ont altéré le message.

L'alphabet source, comme l'alphabet du code, est $\{0, 1\}$. On s'intéresse au codage de messages par blocs : chaque mot de longueur m est codé par un mot de longueur n avec $n \geq m$. Le codage est donc une application de $\{0, 1\}^m$ vers $\{0, 1\}^n$. Parmi les n bits du mot-code que nous allons décrire, m reproduisent le mot-source, les $n - m$ autres sont les bits de correction : le taux de transmission est de $\frac{n}{m}$.

On considère les erreurs comme indépendantes les unes des autres et tous les bits ont la même probabilité d'erreur. Nous nous intéressons donc aux codes correcteurs d'une façon plutôt théorique. En pratique, si on prend un exemple dans les communications sans fil, des problèmes de parasites se posent et l'indépendance des erreurs est compromise.

Pour la suite nous prendrons comme exemple un message qui est un simple caractère encodé en UTF-7. Ce message a donc une taille de 7 bits. Cela égalise ainsi les calculs et l'implémentation pour les tests.

Chapitre 1

Code de Répétition

1.1 Introduction

On transmet simplement plusieurs fois le même message, on double les bits à transmettre pour recevoir deux fois le même message, si les messages reçus ne concordent pas, alors il y a eu une erreur dans la transmission. On peut choisir deux méthodes pour doubler les bits, doubler chaque bit ou doubler le message complet.
Exemple :
101 deviens 110011 si on double chaque bit
101 deviens 101101 si on double le message

1.2 Fiabilité

Une seule erreur peut être détectée à coup sur. Les erreurs ne sont pas detectés dans les cas d'un nombre d'erreur pair se situant au même bits dans les deux messages envoyés.

1.3 Probabilité de détection

Nombre d'erreurs qui ne sont pas détectées (par dénombrement des cas d'indection cités précédement) :

$$\text{Nombre d'erreurs indétectables} = 123$$

Nombre de cas totaux :

$$\text{Nombre de cas} = 2^{(7*2)}$$

Probabilité de détection :

$$P(\text{Détection}) = \frac{\text{Nombre de cas} - \text{Nombre d'erreurs indétectables}}{\text{Nombre de cas}} * 100$$

On obtient alors :

$$P(\text{Détection}) = \frac{2^{(7*2)} - 123}{2^{(7*2)}}$$

$$P(\text{Détection}) \approx 99.25\%$$

1.4 Rendement

Le rendement de ce code est très mauvais, on double au minimum la taille du message :

$$\textit{Rendement} = \text{Taille du message} * \text{Nombre de répétitions}$$

Pour un message sur 7 bits (un simple caractère encodé en UTF-7 par exemple) et avec 1 répétition seulement :

$$\textit{Rendement} = \frac{7}{7*2} = 50\%$$

Pour le même message mais avec 2 répétitions :

$$\textit{Rendement} = \frac{7}{7*3} \approx 33.3\%$$

1.5 Exemple pratique

Voir détail du code en annexe.

```
>> perl envoiNoise.pl 8000 127.0.0.1:9000

 -> envoi du caractere : o

 -> nombre de caracteres envoyes : 100000
```

```

>> perl receptionNoise.pl 9000 127.0.0.1:8000

 -> nombre de receptions : 100000
 -> nombre de receptions supposees bonnes : 18867
 -> nombre de vrais bons caracteres : 18255
 -> nombre d'erreurs au total : 81745
 -> nombre d'erreurs detectees : 81133
 -> nombre d'erreurs non detectees : 612
 -> fiabilitee de l'envoi/reception : 18.255%
 -> fiabilitee de detection d'erreur : 99.2513303565967%
```

Chapitre 2

VRC : Bit de parité

2.1 Introduction

le VRC (Vertical Redundancy Check), plus connu sous le nom de bit de parité, est simplement le rajout d'un bit en fin de message pour assurer la parité du message. Ce dernier bit la valeur nécessaire pour assurer un nombre pair de bit à 1 dans le message final. Il est donc à 0 pour un nombre pair de bit à 1 dans le message de départ, ou est à 1.

2.2 Fiabilité

Une seule erreur peut être détectée à coup sur. Toutes les erreurs où un nombre pair de bits a été modifié ne sont pas détectées, les erreurs détectées sont donc celles où un nombre impair de bits a changé d'état. Si une seule erreur intervient mais porte sur le bit de parité, le message est considéré comme invalide. Ce code ne permet pas la correction d'erreur, il est nécessaire de demander à nouveau l'envoi du message détecté invalide.

2.3 Probabilité de détection

$$P(\text{Transmission Parfaite}) = P(X = 0) = \binom{8}{0} p^0 (1-p)^{8-0} = (1-p)^8$$

$$P(\text{Message Erroné}) = 1 - P(\text{Transmission Parfaite}) = 1 - (1-p)^8$$

$$P(\text{Détection}) = P(1 \text{ erreur}) + P(3 \text{ erreurs}) + P(5 \text{ erreurs}) + P(7 \text{ erreurs})$$

$$= \binom{8}{1} p^1 (1-p)^{8-1} + \binom{8}{3} p^3 (1-p)^{8-3} + \binom{8}{5} p^5 (1-p)^{8-5} + \binom{8}{7} p^7 (1-p)^{8-7}$$

$$= 8p(1-p)^7 + \binom{8}{3} p^3 (1-p)^5 + \binom{8}{5} p^5 (1-p)^3 + 8p^7(1-p)$$

$$P(\text{Reconnaissance Erreur}) = \frac{P(\text{Détection})}{P(\text{Message Erroné})}$$

Pour une probabilité de 10% d'erreurs :

$$P(\text{Transmission Parfaite}) = (1-0.1)^8 = 43\%$$

$$P(\text{Message Erroné}) = 1 - 0.43 = 57\%$$

$$P(\text{Détection}) = 8*0.1(1-0.1)^7 + \binom{8}{3} 0.1^3 (1-0.1)^5 + \binom{8}{5} 0.1^5 (1-0.1)^3 + 8*0.1^7(1-0.1)$$

$$P(\text{Détection}) = 0.8 * 0.9^7 + 56 * 0.1^3 * 0.9^5 + 56 * 0.1^5 * 0.9^3 + 8 * 0.1^7 * 0.9$$

$$P(\text{Détection}) = 42\%$$

$$P(\text{Reconnaissance Erreur}) = \frac{0.42}{0.57} \approx 74\%$$

2.4 Rendement

Le rendement de ce code est très bon :

$$Rendement = \frac{\text{Taille du message}}{\text{Taille du message} + 1}$$

Pour notre message d'exemple (un simple caractère encodé en UTF-7) le rendement est déjà excellent :

$$Rendement = \frac{7}{7+1} = 87.5\%$$

2.5 Exemple pratique

Voir détail du code en annexe.

```
>> perl envoiNoise.pl 9001 127.0.0.1:9000

 -> envoi du caractere : o

 -> nombre de caracteres envoyes : 100000
```

```
>> perl receptionNoise.pl 9000 127.0.0.1:9001

 -> nombre de receptions : 100000
 -> nombre de receptions supposees bonnes : 58311
 -> nombre de vrais bons caracteres : 43079
 -> nombre d'erreurs au total : 56921
 -> nombre d'erreurs detectees : 41689
 -> nombre d'erreurs non detectees : 15232
 -> fiabilitee de l'envoi/reception : 43.079%
 -> fiabilitee de detection d'erreur : 73.2401047065231%
```

Chapitre 3

LRC : Contrôle parité croisée

3.1 Introduction

Le LRC (Longitudinal Redundancy Check) est un mot d'information se composant L caractere de texte. Il s'agit d'un double codage par bits de parité. Les 7 L bits sont rangés dans un tableau de L lignes et 7 colonnes. Chaque ligne est complété par un bit de parité et de même pour les 8 colonnes formés.

Pour rester cohérent avec notre message d'exemple nous prendrons le cas d'une matrice carrée composée de 7 messages de taille 7 chacun. Il sera appliqué horizontalement à la matrice (à chaque message) le bit de parité. La matrice passera donc à une taille de (7+1) sur 8. Le dernier message qui sera généré grâce à l'application du bit de parité verticalement sur la matrice.

3.2 Fiabilité

1 erreur peut être corrigée et jusqu'à trois erreurs peuvent être détectées à coup sûres.

3.3 Probabilité de détection

Probabilité d'exactitude d'un message :

$$P(\text{Exact}) = P(0 \text{ erreur}) + P(1 \text{ erreur})$$

Taille de la matrice $n =$ (Nombre de lignes $+1$)$*$(Longeur du message + Bit de parité)

$$P(0 \text{ erreur}) + P(1 \text{ erreur}) = (1-p)^n + \binom{n}{1} * p * (1-p)^{n-1}$$

Dans notre cas et avec les 10% de chance qu'un bit soit changé :

$$\text{Taille de la matrice } n = 8 * (7+1) = 8^2$$

$$\text{Taille de la matrice } n = 64$$

$$P(0 \text{ erreur}) + P(1 \text{ erreur}) = (1-p)^{64} + 64 * p * (1-p)^{63}$$

$$P(0 \text{ erreur}) + P(1 \text{ erreur}) = (1-0.1)^{64} + 64 * 0.1 * (1-0.1)^{63}$$

$$P(0 \text{ erreur}) + P(1 \text{ erreur}) = (0.9)^{64} + 64 * 0.1 * (0.9)^{63}$$

$$P(0 \text{ erreur}) + P(1 \text{ erreur}) \approx 0.1179\%$$

Un message a donc environ 0.1179% de chance d'être décodé correctement, c'est à dire non détecté faux. Le nombre de cas d'erreur où le LRC est mis à défaut est :

$$\text{Nombre d'indetections} = (49 + 42 + 35 + 28 + 21 + 14 + 7) * 7$$

le nombre de cas total est :

$$P(\text{Nomber de cas}) = 2^{8*(7+1)} = 2^{8*8}$$

$$P(\text{Nombre de cas}) = 2^{64}$$

$$P(\text{Détection}) = \frac{\text{Nombre de cas} - \text{Nombre d'indetections}}{\text{Nombre de cas}} * 100$$

$$P(\text{Détection}) \approx 100\%$$

Comparons la probabilité d'un message decodé à l'aide du LRC à celle de l'information non codé. L'information esr alors transmis directement avec la probabilité de transmission sans erreur. On a :

$$P(\text{n'}) = 7L = 7 * 8 = 56$$

$$P(0) = (1-p)^{n'} = \frac{9}{10} = 0.00273927$$

Calculons alors l'amélioration :

$$P(\text{Amelioration}) = \frac{P(\text{Exact}) - P(0)}{P(0)}$$

$$= \frac{(\frac{9}{10})^{63} * (\frac{64}{10}) - (\frac{9}{10})^{56}}{(\frac{9}{10})^{56}}$$

$$= (\frac{9}{10}^{7} * \frac{64}{10}) - 1 \approx 2.0611$$

On obtient donc une amélioration de 206,11%.

3.4 Rendement

Le rendement de ce code est bon :

$$Rendement = \frac{n * \text{Taille du message}}{(n+1) * (\text{Taille du message} + 1)}$$

Dans le cas que nous étudions (7 messages de 7 bits) :

$$Rendement = \frac{7 * 7}{(7+1) * (7+1)} \approx 76.6\%$$

Chapitre 4

CRC : Code de redondance cyclique

4.1 Introduction

le CRC (Cyclic Redundancy Check), contrôle de redondance cyclique, représente la principale méthode de détection d'erreurs utilisée dans les télécommunications et consiste à protéger des blocs de données, appelés trames. À chaque trame est associé un bloc de données, appelé code ou somme de contrôle (parfois CRC par abus de langage).

On choisit un polynôme générateur, fixé et donc connu des deux entités qui se transmettent le message. Grâce à celui ci, l'émetteur peut générer le code de contrôle qui est le reste de la division avec le message à envoyer. Le récepteur divise ce qu'il a reçut, retrouve le message et sait si il y a eu un problème.

Il existe plusieurs variantes du CRC selon le choix du polynôme : CRC 12, CRC 16, CRC CCIT v41, CRC 32, CRC ARPA.

4.2 Fiabilité

Deux erreurs peuvent être détectées à coup sur grâce au CRC 16. Les erreurs détectées sont seulement celles où un nombre impairs de bit ont changé d'état ou celles qui sont des suites de bit qui ont tous changés (rafales), de taille inférieur au degré du polynôme.
Un code polynomial $C(k, n)$ permet de détecter toutes les erreurs d'ordre $l \leq n - k$ (c'est-à-dire inférieur au degré du polynôme générateur). La probabilité de ne pas détecter les erreurs d'ordre $l > n - k$ est très faible et égale à : $2 - (n - k)$

4.3 Probabilité de détection

Nous sommes finalement arrivé jusqu'à ces calculs et n'avons pas reussi à en comprendre tout les secrets et malices

Dans notre cas :

$$\text{Polynôme CRC 16} = x^{16} + x^{15} + x^2 + 1$$

4.4 Rendement

Le rendement de ce code est dépendant de la taille du message :

$$Rendement = \frac{\text{Taille du message}}{\text{Taille du message} + \text{Degré du polynôme}}$$

Pour notre message d'exemple, un message de seulement 7 bit de longueur, un codage avec du CRC 16 donne un rendement très médiocre :

$$Rendement = \frac{7}{7+16} \approx 30\%$$

Cependant avec un message de taille plus importante comme par exemple un message de 128 bit avec du CRC 16 le rendement devient excellent :

$$Rendement = \frac{128}{128+16} \approx 89\%$$

4.5 Exemple de calcul

Codage Prenons comme exemple un message à envoyer de valeur binaire 0011001 et avec le polynôme CRC 16 = 1100000000000101.
On décale le message de 16 bits vers la gauche 00110010000000000000000.
Ensuite on calcul la somme de contrôle (checksum) de ce message

$$00110010000000000000000|1100000000000101$$

$$xor1100000000010100000$$

$$= 00001000000010100000|1100000000000101$$

$$xor11000000000001010$$

$$= 01000000010101010|1100000000000101$$

$$xor1100000000000101$$

$$= 0100000010101111$$

La somme de contrôle est donc :

$$0100000010101111$$

On concatène cela au message originale à envoyer pour obtenir le message à envoyer :

$$00110010100000010101111$$

Décodage On recois un message de valeur binaire 00110010100000010101111 et avec le polynôme CRC 16 = 1100000000000101. On divise alors ce message par le polynôme pour le vérifier

$$00110010100000010101111|1100000000000101$$

$$xor1100000000000010100000$$

$$= 00001010000000000001111|1100000000000101$$

$$xor1100000000000001010$$

$$= 01100000000000101|1100000000000101$$

$$xor1100000000000101$$

$$= 0000000000000000$$

On s'apercoit que le résultat est nul, aucune erreur n'a été détectée.
On récupère le message en décalant ce que l'on a reçu de 16 bits à droite 0011001

4.6 Exemple pratique

Voir détail du code en annexe.

```
>> perl envoiNoise.pl 9001 127.0.0.1:9000

 -> envoi du caractere : o

 -> nombre de caracteres envoyes : 100000
```

```
>> perl receptionNoise.pl 9000 127.0.0.1:9001

 -> nombre de receptions : 100000
 -> nombre de receptions supposees bonnes : 7557
 -> nombre de vrais bons caracteres : 7301
 -> nombre d'erreurs au total : 92699
 -> nombre d'erreurs detectees : 92443
 -> nombre d'erreurs non detectees : 256
 -> fiabilitee de l'envoi/reception : 7.301%
 -> fiabilitee de detection d'erreur : 99.7238373660989%
```

Conclusion

Nous avons vu au cours de ce dossier différents codes correcteurs utiles dans de nombreux domaines utilisant des transmissions de données. Il existe un grand nombre de méthodes afin de vérifier la validité d'un message à son arrivée. Nous avons donc selectionner les codes correcteurs les moins complexes afin qu'ils soient abordables pour nous. Il est apparut que certains codes simples offraient une très bonne détection aux erreurs mais offraient des rendements très mauvais ce qui posent des problèmes dans des cas de transmission massive de données. Et inversement des codes plus evolués offrent des rendements très mauvais pour des petites transmissions de donnée. Il faut donc prendre tout ces paramètres en compte afin de choisir le type de méthode detection d'erreur à utiliser dans chaque cas particulier. Sur les méthodes que nous avons etudiées il n'existe pas de code parfait. Mais nous avons bien évidemment traité qu'une partie infine du sujet, des codes plus puissants et complexes existent tel que Reed-Solomon par exemple.

Résumé

Différents codes ont été abordées, les moins complexes, jusqu'aux codes polynomiaux. Le code de Répétition étant une simple transmission de n fois le message sont rendement est relativement mauvais et il est très sensible aux erreurs aléatoires régulieres. Le code de Parité est un code qui se sert de la représentation binaire du message pour y ajouter un bit à la fin pour assurer la parité du nombre de bit de poids fort. Il est aussi peu performant, mais cependant très peu couteux en taille puisque occupe qu'un seul bit, son rendement est excellent. Le code de Parité croisé est une amélioration du code de parité grâce à une utisation par bloc de celui-ci. On utilise une matrice de n message précédemment codés et on en calcul verticalement la parité pour créer un message que l'on ajoute comme dernière ligne de cette matrice. Ce code permet de corriger une erreur. Le code de Redondance Cyclique est le plus complexe abordé : il se base sur un polynome diviseur pour générer une somme de contrôle ajoutée en fin de message. Il est presenté dans ce dossier le rendement, la fiabilité et un exemple pratique d'implémentation pour ces codes précédents.

Abstract

It show off some codes into this report, which are the less complex ones, until the polynomial encoding. Repetion code is a simple n time repetitions of the message. Its yield is really poor and this code is very subject to regular random errors. Parity code, Vertical Redundancy Check, use binary representation of the message. A parity bit is generated to ensure parity of significant bit number, and it's added at the end of the message. This one is also weak, but it doesn't use more than only one bit in the message, so its yield is awesome. The Longitudinal Redundancy Check code is an evolution of the previous VRC. Its use some message encoded by the VRC to create a matrix. A new message with parity of vertical reading of the matrix is formed and it's append to this last. This code can resolve one error. Cyclic Redondance Check is the most complex dealt code : it's based on polynomial division of the message to find a checksum which will be append to this message before sending it. Dealt codes are them yield, fiability and an implementation example showed.

Annexe A

Implémentations

A.1 Répétition

Implémentation de l'envoi pour le Code de Répétition

```
#!/usr/bin/perl

use strict;
use Physique::LinkUDP;
use Repetition;

my $link = P_open(@ARGV);
my $nbrRec = 0; # nbr caracteres recus
my $nbrErr = 0; # nbr d'erreurs
my $nbrErrD = 0; # nbr d'erreurs detectees

do{
my $carRec = P_recoitCar($link);

# on test la paritee de cette reception (on test si la reception est
    valide) :
   $nbrErrD+=1 unless (Repetition::verification($carRec)); # la
       reception ne respecte pas la parite

# on test la validite de ce caractere recu :
   $nbrErr+=1 if ( Repetition::decodage($carRec) ne 'o'); # le
       caractere n'est pas celui attendu

   $nbrRec += 1;
}while ($nbrRec < 100000);


print "\n";
print " -> nombre de receptions : ".$nbrRec."\n";
print " -> nombre de receptions supposees bonnes : ".($nbrRec-
    $nbrErrD)."\n";
```

```
28 print " -> nombre de vrais bons caracteres : ".($nbrRec-$nbrErr)."\n
       ";
29 print " -> nombre d'erreurs au total : ".$nbrErr."\n";
30 print " -> nombre d'erreurs detectees : ".$nbrErrD."\n";
31 print " -> nombre d'erreurs non detectees : ".($nbrErr-$nbrErrD)."\n
       ";
32 # attention a la division par "0" ! xD
33 print (" -> fiabilitee de l'envoi/reception : ".(100-100*$nbrErr/
       $nbrRec)."%\n") if($nbrRec != 0);
34 print (" -> fiabilitee de detection d'erreur : ".(100*$nbrErrD/
       $nbrErr)."%\n") if($nbrErr != 0);
35 print "\n";
36
37 P_close($link);
```

Implémentation de la réception pour le Code de Répétition

```
1  #!/usr/bin/perl
2
3  use strict;
4  use Physique::LinkUDP;
5  use Physique::Noise; # pour generer des problemes et erreurs de
       transmission
6  use Repetition;
7
8  my $car = 'o';
9  my $link = P_open(@ARGV);
10 my $nbrEnv = 0;
11 my $temp = Repetition::codage($car); # on ne l'encode qu'une seule
       fois puisque ce caractere ne change pas
12
13 print "\n -> envoi du caractere : ".$car." \n";
14
15 while($nbrEnv<100000)
16 {
17   P_envoiCar($link,$temp);
18   $nbrEnv+=1;
19 }
20
21 print "\n -> nombre de caracteres envoyes : ".$nbrEnv."\n\n";
22
23 P_close($link);
```

Implémentation de la vérification pour le Code de Répétition

```
use strict;
use warnings;

package Repetition;

our $nbrRepetition = 2;

sub codage {
    @_ == 1 or die;
    my ($code) = @_;
    $code = $code x $nbrRepetition;
    return $code;
}

sub verification {
    @_ == 1 or die;
    my ($code) = @_;
    my $chaine1 = substr($code,0,length($code)/$nbrRepetition);
    my $chaine2 = substr($code,length($code)/$nbrRepetition);
    return $chaine1 eq $chaine2;
}

sub decodage {
    @_ == 1 or die;
    my ($code) = @_;
    return substr($code,length($code)/$nbrRepetition);
}

1;
```

A.2 VRC

Implémentation de l'envoi pour le VRC

```
1  #!/usr/bin/perl
2
3  use strict;
4  use Physique::LinkUDP;
5  use Parity;
6
7  sub verification {
8      my $car = ord($_[0]);
9
10 # retour du resultat du test de la parite de ce caractere :
11     return not Parity::parity($car);
12 }
13
14 sub decodage {
15     my $car = ord($_[0]);
16
17 # ou retourner le caractere vide si il y a erreur :
18     return '' unless ( verification(chr($car)) );
19
20 # retourner le decodage du caractere par decalage a droite si la
       paritee est respectee :
21     return chr( $car >> 1);
22 }
23
24
25
26
27 my $link = P_open(@ARGV);
28 my $nbrRec = 0; # nbr caracteres recus
29 my $nbrErr = 0; # nbr d'erreurs
30 my $nbrErrD = 0; # nbr d'erreurs detectees
31
32 do{
33 my $carRec = P_recoitCar($link);
34
35 # on test la paritee de cette reception (on test si la reception est
       valide) :
36    $nbrErrD+=1 unless (verification($carRec)); # la reception ne
         respecte pas la parite
37
38 # on test la validite de ce caractere recu :
39    $nbrErr+=1 if (decodage($carRec) ne 'o'); # le caractere n'est
         pas celui attendu
40
41    $nbrRec += 1;
42 }while ($nbrRec < 100000);
```

```
43
44
45 print "\n";
46 print " -> nombre de receptions : ".$nbrRec."\n";
47 print " -> nombre de receptions supposees bonnes : ".($nbrRec-
       $nbrErrD)."\n";
48 print " -> nombre de vrais bons caracteres : ".($nbrRec-$nbrErr)."\n
       ";
49 print " -> nombre d'erreurs au total : ".$nbrErr."\n";
50 print " -> nombre d'erreurs detectees : ".$nbrErrD."\n";
51 print " -> nombre d'erreurs non detectees : ".($nbrErr-$nbrErrD)."\n
       ";
52 # attention a la division par "0" ! xD
53 print (" -> fiabilitee de l'envoi/reception : ".(100-100*$nbrErr/
       $nbrRec)."%\n") if($nbrRec != 0);
54 print (" -> fiabilitee de detection d'erreur : ".(100*$nbrErrD/
       $nbrErr)."%\n") if($nbrErr != 0);
55 print "\n";
56
57 P_close($link);
```

Implémentation de la réception pour le VRC

```
1  #!/usr/bin/perl
2
3  use strict;
4  use Physique::LinkUDP;
5  use Physique::Noise; # pour generer des problemes et erreurs de
       transmission
6  use Parity;
7
8  sub codage {
9     my ($car) = @_;
10
11 # ajout d'un zero a droite de la valeur binaire du caractere (
       decalage a gauche) :
12    $car = ord($car) << 1;
13
14 # application de la paritee si ce caractere "impair", on inverse le
       dernier bit qui devient donc un "1" :
15    $car = $car ^ 1 if Parity::parity($car);
16
17 # on retourne le caractere et non pas sa valeur binaire :
18    return chr($car);
19 }
20
21
22
23 my $car = 'o';
24 my $link = P_open(@ARGV);
25 my $nbrEnv = 0;
26 my $temp = codage($car); # on ne l'encode qu'une seule fois puisque
       ce caractere ne change pas
27
28 print "\n -> envoi du caractere : ".$car." \n";
29
30 while($nbrEnv<100000)
31 {
32    P_envoiCar($link,$temp);
33    $nbrEnv+=1;
34 }
35
36 print "\n -> nombre de caracteres envoyes : ".$nbrEnv."\n\n";
37
38 P_close($link);
```

Implémentation de la vérification VRC

```
use strict;
use warnings;

package Parity;

sub parity {
    @_ == 1 or die;
    my ($code) = @_;
    my $parity = 0;
    while($code != 0) {
    $parity ^= 1 if ($code & 1);
    $code >>= 1;
    }
    return $parity;
}

1;
```

A.3 CRC

Implémentation de l'envoi pour le CRC

```
1  #!/usr/bin/perl
2
3  use strict;
4  use Physique::LinkUDP;
5  use CRC;
6
7  my $link = P_open(@ARGV);
8  my $nbrRec = 0; # nbr caracteres recus
9  my $nbrErr = 0; # nbr d'erreurs
10 my $nbrErrD = 0; # nbr d'erreurs detectees
11
12 do{
13 my $carRec = P_recoitCar($link);
14
15 # on test la paritee de cette reception (on test si la reception est
       valide) :
16    $nbrErrD+=1 unless (CRC::verification($carRec)); # la reception
        ne respecte pas la parite
17
18 # on test la validite de ce caractere recu :
19    $nbrErr+=1 if (CRC::decodage($carRec) ne 'o'); # le caractere n'
        est pas celui attendu
20
21     $nbrRec += 1;
22 }while ($nbrRec < 100000);
23
24
25 print "\n";
26 print " -> nombre de receptions : ".$nbrRec."\n";
27 print " -> nombre de receptions supposees bonnes : ".($nbrRec-
     $nbrErrD)."\n";
28 print " -> nombre de vrais bons caracteres : ".($nbrRec-$nbrErr)."\n
     ";
29 print " -> nombre d'erreurs au total : ".$nbrErr."\n";
30 print " -> nombre d'erreurs detectees : ".$nbrErrD."\n";
31 print " -> nombre d'erreurs non detectees : ".($nbrErr-$nbrErrD)."\n
     ";
32 # attention a la division par "0" ! xD
33 print (" -> fiabilitee de l'envoi/reception : ".(100-100*$nbrErr/
     $nbrRec)."%\n") if($nbrRec != 0);
34 print (" -> fiabilitee de detection d'erreur : ".(100*$nbrErrD/
     $nbrErr)."%\n") if($nbrErr != 0);
35 print "\n";
36
37 P_close($link);
```

Implémentation de la réception pour le CRC

```
1  #!/usr/bin/perl
2
3  use strict;
4  use Physique::LinkUDP;
5  use Physique::Noise; # pour generer des problemes et erreurs de
       transmission
6  use CRC;
7
8  my $car = 'o';
9  my $link = P_open(@ARGV);
10 my $nbrEnv = 0;
11 my $temp = CRC::codage($car); # on ne l'encode qu'une seule fois
       puisque ce caractere ne change pas
12
13 print "\n -> envoi du caractere : ".$car." \n";
14
15 while($nbrEnv<100000)
16 {
17     P_envoiCar($link,$temp);
18     $nbrEnv+=1;
19 }
20
21 print "\n -> nombre de caracteres envoyes : ".$nbrEnv."\n\n";
22
23 P_close($link);
```

Implémentation de la vérification CRC

```
use strict;
use warnings;
use Digest::CRC ;

package CRC;


sub codage {
    @_ == 1 or die;
    my ($code) = @_;
    my ($check) = unpack "B16", Digest::CRC::crc16($code); # on fixe
         la taille voulue du checksum que l'on calcul a 16bit, cela
        permet de ne pas perdre les 0 devant le premier bit de poids
        fort

#    print Digest::CRC::crc16($code)."\n";
#    print unpack("B16",Digest::CRC::crc16($code))."\n";

    $code.=pack("B16",$check); # on concatene le checksum au message

#    print $code."\n\n";
#    print unpack("B*",$code)."\n\n";

    return $code;
}


sub verification {
    @_ == 1 or die;
    my $temp = $_[0];
    my $check= chop $temp; # on enleve 8 bit de la fin du message
    $check = chop($temp).$check; # on enleve encore 8bits et on y
        concatene les 8bits precedents pour donc retrouver les 16bits
         de fin du message
    $check = unpack "B*", $check;

#    print $temp." > ".unpack("B16",Digest::CRC::crc16($temp))." ?=
    ".$check."\n";

    return ($check eq unpack("B16",Digest::CRC::crc16($temp)));
}


sub decodage {
    @_ == 1 or die;
    my ($code) = unpack "B24",$_[0]; # on creer la chaine qui
        represente la valeur binaire du message
    $code = pack "B8", $code; # recuperation de la partie utile du
```

```
        message qui ne sont que les 8 premiers bits
#    print "decoder : ".$code."\n";
    return $code;
}

1;
```