Présentation des codes redondants cycliques (CRC) & Illustration et limites du Code Hamming

Réalisé par les élèves ingénieurs

AHOUANDJINOU Bill Dieumène Michaël & ALONOUMI Cédric Bosco

Ecole Polytechnique d'Abomey-Calavi – Université d'Abomey-Calavi

EPAC – UAC

Département de Génie Informatique et Télécommunications

25/04/2022

Table des matières

Section introductrice

Le codage binaire est très pratique pour une utilisation dans des appareils électroniques tels qu'un ordinateur, dans lesquels l'information peut être codée grâce à un signal électrique. Cependant le signal électrique peut subir des perturbations (distortion, présence de bruit), notamment lors du transport des données sur un long trajet. Ainsi, le contrôle de la validité des données est nécessaire pour certaines applications (professionelles, bancaires, industrielles, confidentielles, relatives à la sécurité, ...). C'est pourquoi il existe des mécanismes permettant de garantir un certain niveau d'intégrité des données, c'est-à-dire de fournir au destinataire une assurance que les données reçues sont bien similaires aux données émises. La protection contre les erreurs peut se faire de deux façons :
- soit en *fiabilisant le support de transmission*, c'est-à-dire en se basant sur une protection physique.
- soit en *mettant en place des mécanismes logiques de détection et de correction des erreurs*.
Nous n'aborderons ce document que ce dernier aspect.

Chapitre 1

Code CRC

A Contrôle d'erreur : Généralités

Des systèmes de plus en plus performants ont été mis au point pour perfectionner la détection d'erreurs de transmission. Ces codes sont généralement appelés codes **autocorrecteurs** ou **autovérificateurs**.

Une technique simple de détection d'erreurs est appelée le **contrôle de parité** [1]. Nous avons également le **contrôle de parité croisé** [2] qui consiste non pas à contrôler l'intégrité des données d'un caractère, mais à contrôler l'intégrité des bits de parité d'un bloc de caractères.

La plupart des systèmes de contrôle d'erreur au niveau logique sont basés sur un ajout d'information [3] permettant de vérifier la validité des données. Une vérification plus sophistiquée que la parité peut donc être effectuée en associant des informations de contrôle supplémentaires à la fin de chaque bloc de données transmis. On appelle **somme de contrôle** [4] cette information supplémentaire. Elle est calculée initialement par l'expéditeur et vérifiée par le destinataire. L'une des méthodes standard de calcul d'une somme de contrôle est appelée **Contrôle de Redondance Cyclique** (CRC).

1. appelé parfois VRC, pour Vertical Redundancy Check ou Vertical Redundancy Checking
2. aussi appelé contrôle de redondance longitudinale ou Longitudinal Redundancy Check et noté LRC
3. on parle de « redondance »
4. en anglais checksum

B Contrôle de Redondance Cyclique

Le contrôle de redondance cyclique, noté CRC, ou en anglais Cyclic Redundancy Check est un moyen de contrôle d'intégrité des données puissant et facile à mettre en œuvre. Il représente la principale méthode de détection d'erreurs utilisée dans les télécommunications.

Un CRC est appelé CRC à n bits lorsque sa valeur de contrôle a une longueur de n bits. Pour un n donné , plusieurs CRC sont possibles, chacun avec un polynôme différent. Un tel polynôme a le degré le plus élevé n, ce qui signifie qu'il a $n+1$ termes. En d'autres termes, le polynôme a une longueur de $n+1$. Son encodage nécessite donc $n+1$ bits. Notez que la plupart des spécifications polynomiales suppriment le MSB [5] ou le LSB [6], car ils sont toujours à 1. Le système de détection d'erreurs le plus simple, le bit de parité , est en fait un CRC à 1 bit : il utilise le polynôme générateur $x+1$ (deux termes), et porte le nom de CRC-1.

B.1 Principe

Le contrôle de redondance cyclique consiste à protéger des blocs de données, appelés trames (*frames* en anglais). A chaque trame est associé un bloc de données, appelé code de contrôle. Le code CRC contient des éléments redondants vis-à-vis de la trame, permettant de détecter les erreurs.

Le principe du CRC consiste à traiter les séquences binaires comme des polynômes binaires, c'est-à-dire des polynômes dont les coefficients correspondent à la séquence binaire. Ainsi la séquence binaire 110101001 peut être représentée sous la forme polynomiale suivante : $X^8+X^7+X^5+X^3+1$. Une séquence de n bits constitue donc un polynôme de degré maximal $n-1$.

Dans ce mécanisme de détection d'erreur, un polynôme prédéfini appelé polynôme **générateur** et noté $G(X)$ est connu de l'émetteur et du récepteur. La détection d'erreur consiste pour l'émetteur à effectuer un algorithme sur les bits de la trame afin de générer un CRC, et de transmettre ces deux éléments au récepteur. Il suffit alors au récepteur d'effectuer le même calcul afin de vérifier que le CRC est valide. L'appareil peut prendre des mesures correctives, telles que relire le bloc ou demander qu'il soit à nouveau envoyé. Sinon, les données

5. Most Significant Bit
6. Low Significant Bit

sont supposées être exemptes d'erreurs (bien que, avec une faible probabilité, elles puissent contenir des erreurs non détectées ; ceci est inhérent à la nature de la vérification des erreurs).

B.2 Procédure de calcul

Soit M le message correspondant aux bits de la trame à envoyer et $M(X)$ le polynôme associé.
Appelons M' le message transmis, c'est-à-dire le message initial auquel aura été concaténé le CRC de n bits.
Le CRC est tel que $M'(X)/G(X) = 0$. Le code CRC est ainsi égal au reste de la division polynomiale de $M(X)$ (auquel on a préalablement concaténé n bits nuls correspondant à la longueur du CRC) par $G(X)$.
La mise en garde importante est que les coefficients polynomiaux sont calculés selon l'arithmétique d'un corps fini, de sorte que l'opération d'addition peut toujours être effectuée en parallèle au niveau du bit (il n'y a pas de report entre les chiffres). Autrement dit, il s'agira d'utiliser l'opérateur XOR.

B.3 Application

Soit M le message correspondant aux bits de la trame à envoyer et $M(X)$ le polynôme associé. Appelons M' le message transmis et $G(X)$, le polynôme générateur associé au générateur G. Prenons le message M de 16 bits suivant : 1011000100101010 (noté $B12A$ en hexadécimal).
Prenons $G(X) = X^3 + 1$ (représenté en binaire par 1001). Etant donné que $G(X)$ est de degré 3, il s'agit d'ajouter 3 bits nuls à M, soit : 1011000100101010000.
Le CRC étant égal au reste de la division de M par G, en prenant soin d'appliquer l'opérator logique XOR à la place de la soustraction, on retrouve $CRC = 001$.
Pour créer M' il suffit de concaténer le CRC ainsi obtenu aux bits de la trame à transmettre : $M' = 1011000100101010001$. Ainsi, si le destinataire du message effectue la division de M' par G, en respectant la même règle concernant l'opérateur XOR, il obtiendra un reste nul si la transmission s'est effectuée sans erreur.
Etant donné que le quotient ne nous est pas très utile dans cette application, on peut procéder directement à une série d'opérations avec XOR entre M auquel on concatène 3 bits à 0 (lon-

gueur du CRC). Cette série d'opérations est effectuée du MSB vers le LSB en s'assurant que chaque opération s'effectue entre même nombre de bits.

B.4 Implémentation

Le code C++ ci-après permet de calculer le CRC à partir de la donnée du message et du générateur.

```
# include <iostream>
using namespace std;

int main()
{
    string msg, crc, encoded = "";
    cout << "Message = ";
    getline(cin, msg);
    cout << "Generator = ";
    getline(cin, crc);
    int m = msg.length(), n = crc.length();
    encoded += msg;
    for(int i = 1; i<= n-1; i++)
    encoded += '0';
    cout << 'encoded = ' << ' ' + encoded << '\n'
    for(int i = 0; i <= encoded.length()-n; )
    {
        for(int j = 0; j < n; j++)
            encoded[i+j] = encoded[i+j] == crc[j]? '0':'1';
        for(;i<encoded.length() && encoded[i] != '1'; i++);
    }
    cout << "CRC = " << encoded.substr(encoded.length()-n+1) << "\n";
    cout << "Message sent = " << msg + ' ' + encoded.substr(encoded.lengt

    return 0;
}
```

Le code C++ ci-après permet de vérifier si le message reçu par le récepteur est sans erreur ou non à partir de la donnée du message reçu et du générateur.

```
# include <iostream>
using namespace std;
int main()
{
    string crc, encoded;
    cout << "Received Message = ";
    getline(cin, encoded);
    cout << "Generator = ";
    getline(cin, crc);
    for(int i = 0; i <= encoded.length() - crc.length();)
    {
        for(int j = 0; j < crc.length(); j++)
            encoded[i+j] = encoded[i+j] == crc[j]? '0' : '1';
        for( ; i < encoded.length() && encoded[i] != '1'; i++);
    }
    for(char i : encoded.substr(encoded.length() - crc.length()))
        if(i != '0')
        {
            cout << "Error in communication ..." << "\n";
            return 0;
        }
    cout << "No error !" << "\n";
    return 0;
}
```

B.5 Intégrité des données

Les CRC sont spécifiquement conçus pour protéger contre les types courants d'erreurs sur les canaux de communication, où ils peuvent fournir une assurance rapide et raisonnable de l'intégrité des messages livrés. Cependant, ils ne sont pas adaptés à la protection contre l'altération intentionnelle des données.

- Premièrement, comme il n'y a pas d'authentification, un attaquant peut éditer un message et recalculer le CRC sans que la substitution ne soit détectée. Lorsqu'ils sont stockés avec les données, les CRC et les fonctions de hachage cryptographique ne protègent pas en eux-mêmes contre la modification intentionnelle des données. Toute application nécessitant une protection contre de telles attaques doit utiliser des mécanismes d'authentification cryptographique, tels que des codes d'authentification de message ou des signatures numériques (qui sont généralement basées sur des fonctions de hachage cryptographique).
- Deuxièmement, contrairement aux fonctions de hachage cryptographique, le CRC est une fonction facilement réversible, ce qui le rend impropre à une utilisation dans les signatures numériques.
- Troisièmement, CRC est une fonction linéaire, en conséquence, même si le CRC est chiffré avec un chiffrement de flux qui utilise XOR comme opération de combinaison, à la fois le message et le CRC associé peut être manipulé sans connaître la clé de chiffrement. C'était l'un des défauts de conception bien connus du protocole Wired Equivalent Privacy(WEP) [7].

C Polynômes générateurs

Les polynômes générateurs les plus couramment employés sont :

CRC-12 : $X^{12}+X^{11}+X3+X2+X+1$

CRC-16 : $X^{16}+X^{15}+X^2+1$

CRC CCITT V41 [8] : $X^{16}+X^{12}+X^5+1$

CRC-32 (Ethernet) : $X^{32}+X^{26}+X^{23}+X22+X16+X12+X11+X10+X^8+X^7+X^5+X^4+X^2+X+1$

CRC ARPA : $X^{24}+X^{23}+X^{17}+X^{16}+X^{15}+X^{13}+X^{11}+X^{10}+X^9+X^8+X^5+X^3+1$

7. WEP is a est un protocole de sécurité, spécifié par le standard IEEE Wireless Fidelity (Wi-Fi) 802.11b. Ce standard a été conçu pour permettre la mise sur pied d'un réseau LAN sans fil avec un niveau de sécurié et de confidentialité comparable à celui d'un réseau LAN filaire

8. Ce code est notamment utilisé dans la procédure HDLC

C.1 Concevoir des polynômes

La sélection du polynôme générateur est la partie la plus importante de la mise en œuvre de l'algorithme CRC. Le polynôme doit être choisi pour maximiser les capacités de détection d'erreurs tout en minimisant les probabilités de collision globales. L'attribut le plus important du polynôme est sa longueur en raison de son influence directe sur la longueur de la valeur de contrôle calculée. Les longueurs polynomiales les plus couramment utilisées sont :

- 9 bits (CRC-8)
- 17 bits (CRC-16)
- 33 bits (CRC-32)
- 65 bits (CRC-64)

La conception du polynôme CRC dépend de la longueur totale maximale du bloc à protéger (données + bits CRC), des fonctionnalités de protection contre les erreurs souhaitées et du type de ressources pour la mise en œuvre du CRC, ainsi que des performances souhaitées. Une idée fausse commune est que les « meilleurs » polynômes CRC sont dérivés soit de polynômes irréductibles, soit de polynômes irréductibles multipliés par le facteur $1+X$, ce qui ajoute au code la capacité de détecter toutes les erreurs affectant un nombre impair de bits. En réalité, tous les facteurs décrits ci-dessus doivent entrer dans la sélection du polynôme et peuvent conduire à un polynôme réductible. Cependant, le choix d'un polynôme réductible entraînera une certaine proportion d'erreurs manquées, en raison de l'anneau quotient ayant zéro diviseur.

Chapitre 2

Code Hamming

A Présentation

L'objectif d'un code correcteur est la détection et la correction d'erreurs après la transmission d'un message. Cette correction est permise grâce à l'ajout d'informations redondantes. Le message est plongé dans un ensemble plus grand, la différence de taille contient la redondance, l'image du message par le plongement est transmise. En cas d'altération du message, la redondance est conçue pour détecter ou corriger les erreurs. Un **code de Hamming** suit cette logique, la redondance permet exactement la correction d'une altération sur une unique lettre du message. Un code de Hamming est ainsi un code correcteur linéaire. Il permet la détection et la correction automatique d'une erreur si elle ne porte que sur une lettre du message.

B Fonctionnement

Soit M le message à transmettre de taille m.

Soit N le message transmis de taille n.

Soit K le mot contenant les bits de redondance de taille k.

Condition : k et m sont tels que $2^k \geq m+k+1$

Au niveau de l'émetteur :

- **Position des bits de redondance :** On code chaque bloc de m bits de données par un bloc de $n = 2^k - 1$ bits en ajoutant donc k bits, dits de correction, à certaines positions

au bloc de m bits. Les k bits de correction sont placés dans le bloc envoyé aux positions d'indice de puissance de 2 en comptant à partir du bit le moins significatif.

Exemple :

Pour $m = 4$, on a $k = 3$ et $n = 7$. $M = m_4m_3m_2m_1$ et $K = k_3k_2k_1$. On obtient $N = m_4m_3m_2k_3m_1k_2k_1$.

- **Calcul des bits de redondance :** Les k bits de correction sont calculés en utilisant une matrice de parité H de taille $k \times n$ déterminée en fonction des numéros des bits contrôlés par chaque bit de redondance dit également bit de parité. Pour déterminer cette matrice, on procède de la façon suivante :

– Écrivez les positions des bits de N à partir de 1 sous forme binaire (1, 10, 11, 100, etc.). On rappelle que toutes les positions de bit qui sont une puissance de 2 sont marquées comme bits de parité (1, 2, 4, 8, etc.) et toutes les autres positions de bit sont marquées comme bits de données.

  * Le bit de parité 1 couvre toutes les positions de bits dont la représentation binaire comprend un 1 à la position la moins significative (1, 3, 5, 7, 9, 11, etc.).
  * Le bit de parité 2 couvre toutes les positions de bits dont la représentation binaire comprend un 1 en deuxième position à partir du bit le moins significatif (2, 3, 6, 7, 10, 11, etc.).
  * Le bit de parité 4 couvre toutes les positions de bits dont la représentation binaire comprend un 1 en troisième position à partir du bit le moins significatif (4–7, 12–15, 20–23, etc.).
  * Le bit de parité 8 couvre toutes les positions de bits dont la représentation binaire comprend un 1 en quatrième position à partir des bits les moins significatifs (8–15, 24–31, 40–47, etc.).
  * En général, chaque bit de parité couvre tous les bits dont la position, en binaire, a un bit égal à 1 à la position d'indice égal au numéro du bit de parité.

– Dans H, chaque ligne contient les informations concernant les bits contrôlés ou non par le bit de parité, c'est-à-dire que si par exemple le 6e bit de N est contrôlé par le $2e$ bit de parité, alors à la $2e$ ligne et $6e$ colonne de la matrice H, on aura la valeur 1, ou 0 si le bit de parité considéré ne contrôle pas le $6e$ bit de N. Le produit de H par la matrice colonne définie par N donne la matrice colonne K contenant les valeurs des différents bits de parité. On notifie que les opérations d'addition sont effectuées à modulo 2.

Tout de même, on peut faire simple en considérant les valeurs des bits contrôlés dans N

par le bit de parité et appliquer le principe de contrôle de parité. Puisque nous vérifions la parité paire, on définit un bit de parité sur 1 si le nombre total de bit à 1 dans les positions qu'il vérifie est impair et sur 0 si le nombre total de bit à 1 dans les positions qu'il vérifie est pair.

Après détermination des k bits de redondance, on envoie le bloc N.

Au niveau du récepteur :

Le récepteur reçoit un bloc Q qui est égal à N s'il n'y a pas d'erreur.

Pour la détection d'une erreur, on fait le produit matriciel de H par la matrice colonne générée à partir des bits du bloc Q qui donne la matrice C.

C donne la valeur de la position de l'éventuel bit erroné. Si C est la matrice nulle, alors il n'y a pas d'erreur dans le message reçu. Autrement, l'erreur se situe à la position indiquée par cette matrice. Il suffira juste dans ce cas d'inverser la valeur binaire du bit se trouvant à cette position pour le corriger.

C Application

Au niveau de l'émetteur :

On considère l'exemple de la sous-section B : Pour $m = 4$, on a $k = 3$ et $n = 7$. $M = m_4m_3m_2m_1$ et $K = k_3k_2k_1$. On obtient $N = m_4m_3m_2k_3m_1k_2k_1$.

k_1 contrôle les bits d'indice 1, 3, 5 et 7 k_2 contrôle les bits d'indice 2, 3, 6 et 7 k_3 contrôle les bits d'indice 4, 5, 6 et 7 Ainsi on a :

$$H = \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}$$

et

$$N = \begin{pmatrix} k_1 \\ k_2 \\ m_1 \\ k_3 \\ m_2 \\ m_3 \\ m_4 \end{pmatrix}$$

Le produit donne :

$$K = \begin{pmatrix} k_1 \\ k_2 \\ k_3 \end{pmatrix}$$

En prenant $M = 1010$, on a :

$$N = \begin{pmatrix} k_1 \\ k_2 \\ 0 \\ k_3 \\ 1 \\ 0 \\ 1 \end{pmatrix}$$

$$H \times N = K = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$$

D'où le message envoyé est : $N = 1010010$.

Au niveau du récepteur :

Il reçoit $Q = q_7q_6q_5q_4q_3q_2q_1$.

$H \times Q = C$

Supposons que N est reçu sans erreur. On a donc $N = Q = 1010010$ et C est la matrice nulle.

On suppose à présent qu'il y a une erreur au bit d'indice 5. Donc $Q = 1000010$.

$H \times Q$ donne

$$C = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}$$

La conversion en décimale de $C = 101$ (en binaire) donne $C = 5$ (en décimal) ce qui correspond à notre hypothèse de départ. Il suffira donc juste d'inverser la valeur du 5^e bit à partir du LSB pour corriger l'erreur.

D Implémentation

Le code en langage python ci-dessous génère un GUI [1] permettant de simuler le fonctionnement d'un code Hamming (voir figure 2.1).

```
# Author : Bill AHOUANDJINOU
# April 2022

import gi
gi.require_version('Gtk', '3.0')
from gi.repository import Gtk
from hamming import *

def Sent(button, entryData, labelS) :
    data = entryData.get_text()
    m = len(data)
    k = calcRedundantBits(m)
    sentData = posRedundantBits(data, k)
    sentData = calcParityBits(sentData, k)
    print(f'Sent Data = {sentData}')
    labelS.set_text(str(sentData))

def Received(button,entryData, labelR) :
    data = entryData.get_text()
    m = len(data)
    k = calcRedundantBits(m)
    sentData = posRedundantBits(data, k)
    sentData = calcParityBits(sentData, k)
    print(f'Sent Data = {sentData}')
    receivedData = entryData.get_text()
    print(f"Received Data is {receivedData}")
    position = detectError(receivedData, k)
    if position == 0 :
```

1. Graphical User Interface

```
            print("There is no error in this transmission.")
            labelR.set_text("No error !")
        else :
            a = "Error in bit at position " + str(position)
            labelR.set_text(a)
        print(f"The position of error is {position}.")

def Clear(button, labelTM, labelPosition, entryData, entryMessage) :
    entryData.set_text('')
    entryMessage.set_text('')
    labelTM.set_text('Transmitted data')
    labelPosition.set_text('Error position')

def Hamming() :
    window = Gtk.Window()
    window.set_border_width(40)
    window.connect('delete-event', Gtk.main_quit)

    grid = Gtk.Grid()

    labelS = Gtk.Label(label = "Sender")
    entryData = Gtk.Entry()
    generate = Gtk.Button(label = "Generate TM")
    labelTM = Gtk.Label(label = "Transmitted Data")
    grid.attach(labelS, 0, 0, 10, 1)
    grid.attach(entryData, 2, 2, 3, 1)
    grid.attach(generate, 2, 4, 3, 1)
    grid.attach(labelTM, 2, 6, 3, 1)

    labelR = Gtk.Label(label = "Receiver")
    entryMessage = Gtk.Entry()
    control = Gtk.Button(label = "Control Transmission")
    labelPosition = Gtk.Label(label = "Error Position")
```

```
    grid.attach(labelR, 0, 9, 10, 2)
    grid.attach(entryMessage, 2, 11, 3, 1)
    grid.attach(control, 2, 13, 3, 1)
    grid.attach(labelPosition, 2, 15, 3, 1)

    clear = Gtk.Button(label = "Clear all")
    grid.attach(clear, 2, 17, 3, 1)
    clear.connect('clicked', Clear, labelTM, labelPosition, entryData, en

    generate.connect('clicked', Sent, entryData, labelTM)
    control.connect('clicked', Received, entryMessage, labelPosition)

    window.add(grid)
    window.show_all()
    Gtk.main()

Hamming()
```

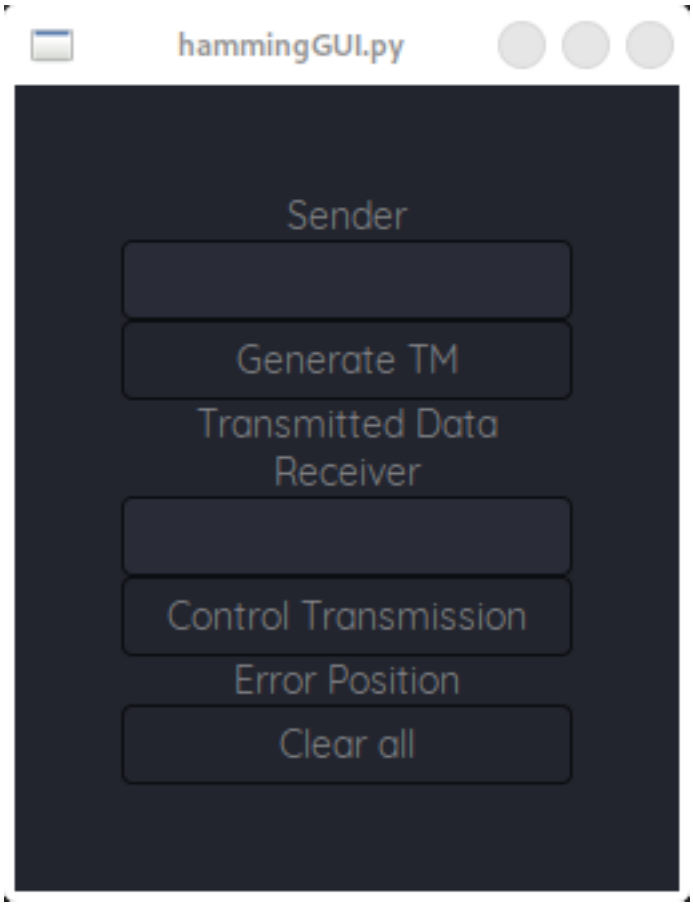

FIGURE 2.1 : Interface GUI de simulation du code Hamming

Par ailleurs, il est possible de simuler une transmission numérique en utilisant le code Hamming grâce au logiciel Matlab. En tapant dans le *command window* la commande *doc_hamming* nous voyons afficher la version complète du modèle de simulation utilisant le code Hamming. A la première version, nous ajoutons quelques oscilloscopes et nous obtenons la figure 2.2.

E Limites du code Hamming

La limite principale du code Hamming (n,m,k) repose sur sa capacité de correction. Le troisième paramètre du code (k) indique la distance de Hamming maximale entre deux mots

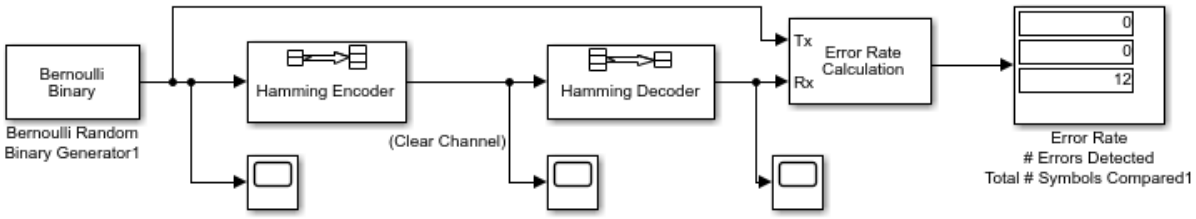

FIGURE 2.2 : Modèle de simulation d'une transmission numérique utilisant le code Hamming avec Matlab

de code [2].

La capacité de correction du code de Hamming est donc de $\frac{k-1}{2}$ erreur(s) pour n bits de données. Le code Hamming (7,4,3) utilisé dans la sous-section C permet donc de ne corriger qu'une erreur pour 7 bits. En revanche, elle peut détecter de 2 erreurs au plus. La valeur de C correspond donc à la somme (modulo 2) des valeurs des colonnes de H portant indice les numéros des deux bits erronnés. Au delà de 2, la détection est incertaine.

2. La distance de Hamming correspond au nombre de bits différents entre deux mots binaires (elle n'existe que si les deux mots sont de longueur égale)

Conclusion

Il est très probable de recevoir des messages erronnés après une transmission. Nous pouvons, de ce fait, prendre des précautions de traitement avant l'envoi pour essayer de détecter ces éventuelles erreurs en utilisant des codes détecteurs (code CRC) et/ou correcteurs (code Hamming) d'erreurs. Malgré l'efficacité de ces codes, ils ont des limites qui restreignent donc leur usage.

Annexes

Présentation des résultats d'application

On considère l'application développée plus haut.

- **Code de la section B.4 :**

  Génération du code CRC à l'émission :

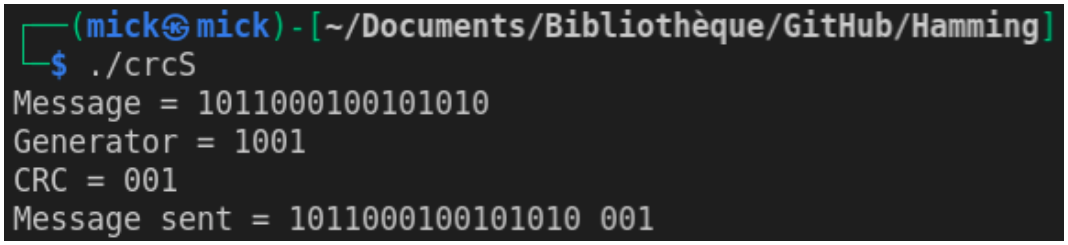

FIGURE 2.3 : Génération du code CRC à l'émission

Vérification de présence d'erreur dans le message reçu au niveau du récepteur :

FIGURE 2.4 : Vérification d'erreur dans le message reçu : Transmission sans erreur

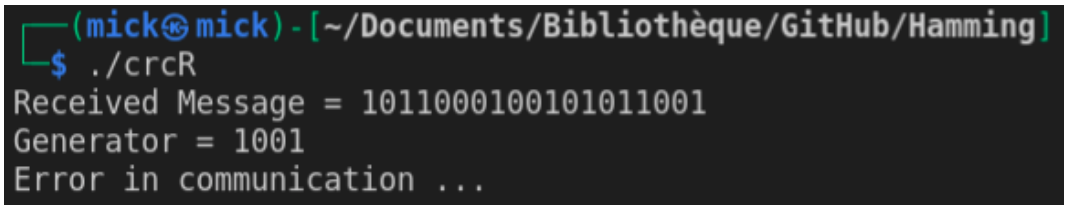

FIGURE 2.5 : Vérification d'erreur dans le message reçu : Transmission erronnée

- **Code de la section D :**

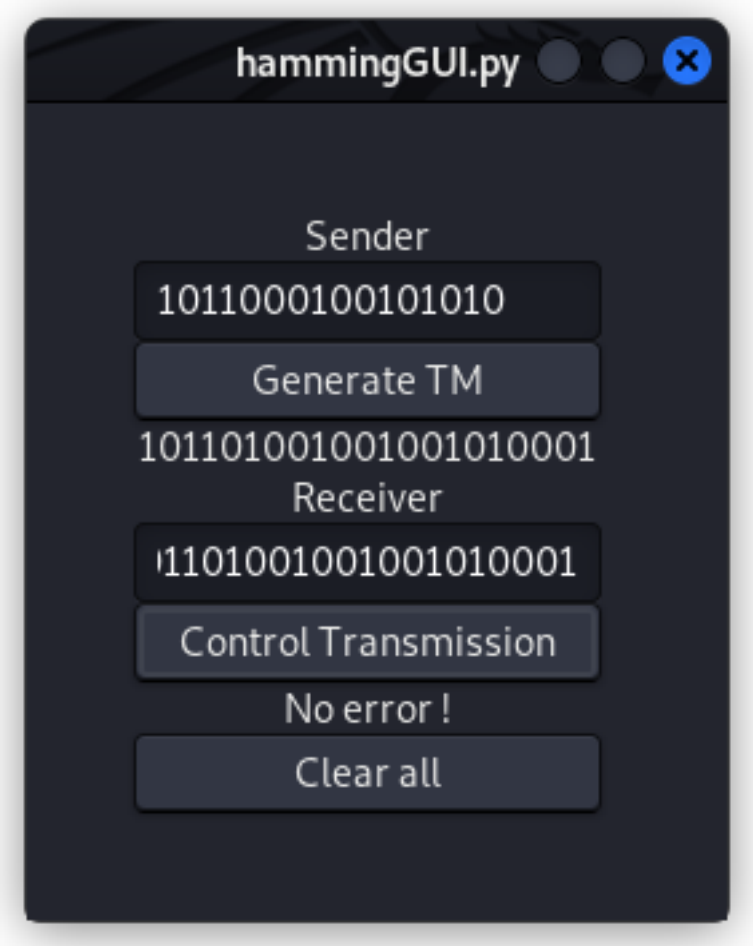

FIGURE 2.6 : Application du code Hamming : transmission sans erreur

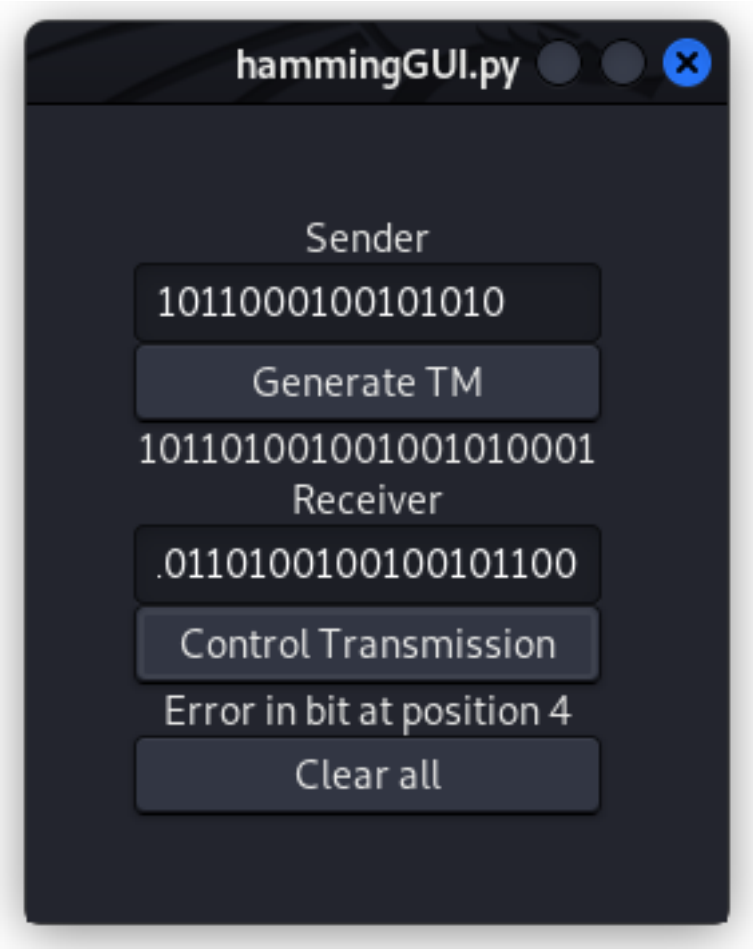

FIGURE 2.7 : Application du code Hamming : transmission erronnée

Table des figures

Bibliographie

- https://www.commentcamarche.net/contents/97-controle-d-erreur-crc, consulté en avril 2022
- https://waytolearnx.com/2019/06/techniques-de-detection-derreur.html, consulté en avril 2022
- https://www.techno-science.net/glossaire-definition/Code-de-Hamming-7-4.html, consulté en avril 2022
- https://www.mathworks.com/help/comm/ug/error-detection-and-correction.html, consulté en avril 2022