Somme de contrôle

Table des matières

1. Introduction....2
2. Contrôle de parité....2
3. Contrôle de parité croisé....3
4. Le contrôle de redondance cyclique....3
4.1. Principe....4
4.2. Application pratique....4
4.3. Polynômes générateurs....5
5. Exemples de calcul de checksum....5
5.1. Empreinte sur 2 octets....5
5.2. Hachage....6
5.3. Complément à 255....6

La somme de contrôle (checksum en anglais), parfois appelée « empreinte », est un nombre qu'on ajoute à un message à transmettre pour permettre au récepteur de vérifier que le message reçu est bien celui qui a été envoyé. L'ajout d'une somme de contrôle à un message est une forme de contrôle par redondance.

1. Introduction

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 la présence ou non d'un signal électrique.

Cependant le signal électrique peut subir des perturbations (distorsion, 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 (professionnelles, 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**. Une liaison conventionnelle a généralement un taux d'erreur compris entre 10^{-5} et 10^{-7}.
- soit en mettant en place des mécanismes **logiques** de *détection* et de *correction* des erreurs.

La plupart des systèmes de contrôle d'erreur au niveau logique sont basés sur un ajout d'information (on parle de « redondance ») permettant de vérifier la validité des données. On appelle **somme de contrôle**[1] cette information supplémentaire.

C'est ainsi que des systèmes de détection d'erreur plus perfectionnés ont été mis au point, ces codes sont appelés :

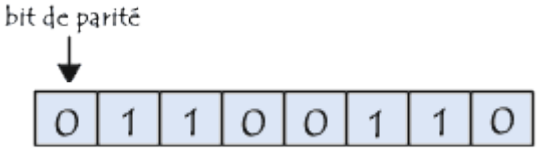
- codes autocorrecteurs
- codes autovérificateurs

2. Contrôle de parité

Le contrôle de parité (appelé parfois *VRC*[2]) est un des systèmes de contrôle les plus simples.

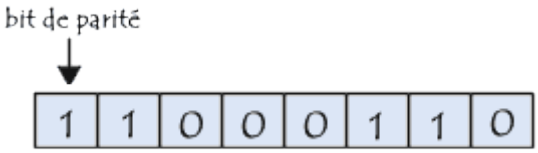
Il consiste à ajouter un bit supplémentaire (appelé **bit de parité**) à un certain nombre de bits de données appelé *mot de code* (généralement 7 bits, pour former un octet avec le bit de parité) dont la valeur (*0* ou *1*) est telle que le nombre total de bits à 1 soit pair. Pour être plus explicite il consiste à ajouter un *1* si le nombre de bits du mot de code est impair, *0* dans le cas contraire.

Prenons l'exemple suivant :

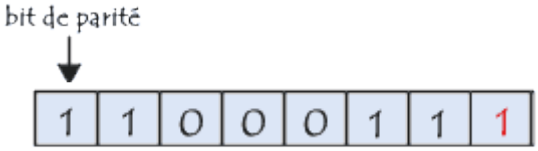


Dans cet exemple, le nombre de bits de données à *1* est pair, le bit de parité est donc positionné à *0*. Dans l'exemple suivant, par contre, les bits de données étant en nombre impair, le bit de parité est à 1 :

1 Checksum en anglais
2 Vertical Redundancy Checking

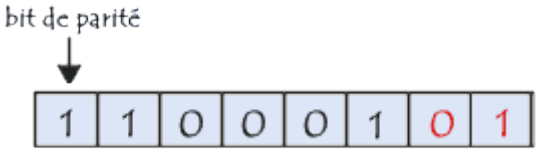


Imaginons désormais qu'après transmission le bit de poids faible (le bit situé à droite) de l'octet précédent soit victime d'une interférence :



Le bit de parité ne correspond alors plus à la parité de l'octet : **une erreur est détectée**.

Toutefois, si deux bits (ou un nombre pair de bits) venaient à se modifier simultanément lors du transport de données, aucune erreur ne serait alors détectée...



Le système de contrôle de parité ne détectant que les erreurs en nombre impair, il ne permet donc de détecter que 50% des erreurs.

Ce système de détection d'erreurs possède également l'inconvénient majeur de ne pas permettre de corriger les erreurs détectées (le seul moyen est d'exiger la retransmission de l'octet erroné...).

3. Contrôle de parité croisé

Le contrôle de parité croisé (aussi appelé **LRC**[3]) 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.

Soit « HELLO » le message à transmettre, en utilisant le code ASCII standard. Voici les données telles qu'elles seront transmises avec les codes de contrôle de parité croisé :

Lettre	***Code ASCII (sur 7 bits)***	***Bit de parité (LRC)***
H	1001000	0
E	1000101	1
L	1001100	1
L	1001100	1
0	1001111	1
VRC	1000010	0

4. Le contrôle de redondance cyclique

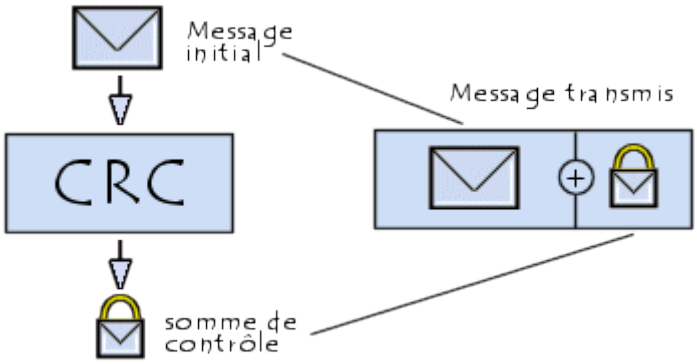
Le **contrôle de redondance cyclique** (noté **CRC**[4]) est un moyen de contrôle d'intégrité des données

3 Longitudinal Redundancy Check : contrôle de redondance longitudinale
4 Cyclic Redundancy Check

puissant et facile à mettre en œuvre. Il représente la principale méthode de détection d'erreurs utilisée dans les télécommunications.

4.1. Principe

Le **contrôle de redondance cyclique** consiste à protéger des blocs de données, appelés *trames*[5]. A chaque trame est associé un bloc de données, appelé *code de contrôle* (parfois *CRC* par abus de langage ou *FCS*[6] dans le cas d'un code de 32 bits). Le *code CRC* contient des éléments redondants vis-à-vis de la trame, permettant de détecter les erreurs, mais aussi de les réparer.



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 *0110101001* peut être représentée sous la forme polynomiale suivante :

$$0*X^9 + 1*X^8 + 1*X^7 + 0*X^6 + 1*X^5 + 0*X^4 + 1*X^3 + 0*X^2 + 0*X^1 + 1*X^0$$

soit

$$X^8 + X^7 + X^5 + X^3 + X^0 = X^8 + X^7 + X^5 + X^3 + 1$$

De cette façon, le bit de poids faible de la séquence (le bit le plus à droite) représente le degré 0 du polynôme ($X^0 = 1$), le 4ème bit en partant de la droite représente le degré 3 du polynôme (X^3)... Une séquence de *n* bits constitue donc un polynôme de degré maximal *n-1*. Toutes les expressions polynomiales sont manipulées par la suite avec une arithmétique modulo 2.

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.

4.2. Application pratique

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).

Le plus simple est encore de prendre un exemple : prenons le message *M* de 16 bits suivant : *1011*

5 frames en anglais
6 Frame Check Sequence

0001 0010 1010 (noté *B1* en hexadécimal). Prenons G(X) = $X^3 + 1$ (représenté en binaire par *1001*).

Étant donné que G(X) est de degré *3*, il s'agit d'ajouter *4* bits nuls à M : *10110001001010100000*.

Le CRC est égal au reste de la division de M par G.

Pour créer M' il suffit de concaténer le CRC ainsi obtenu aux bits de la trame à transmettre :

M' = 1011000100101010 + 0011
M' = 10110001001010100011

Ainsi, si le destinataire du message effectue la division de M' par G, il obtiendra un reste nul si la transmission s'est effectuée sans erreur.

4.3. Polynômes générateurs

Les polynômes générateurs les plus couramment employés sont :

- **CRC-12** : $X^{12} + X^{11} + X^3 + X^2 + X + 1$
- **CRC-16** : $X^{16} + X^{15} + X^2 + 1$
- **CRC CCITT V41** : $X^{16} + X^{12} + X^5 + 1$

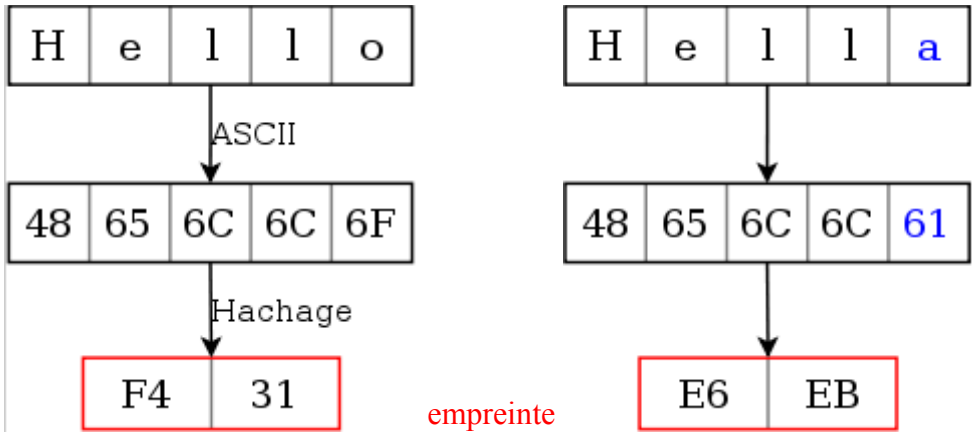
(Ce code est notamment utilisé dans la procédure *HDLC*[7])

- **CRC-32 (Ethernet)** : $X^{32} + X^{26} + X^{23} + X^{22} + X^{16} + X^{12} + X^{11} + X^{10} + X^8 + X^7 + X^5 + X^4 + X^2 + X + 1$
- **CRC ARPA** : $X^{24} + X^{23} + X^{17} + X^{16} + X^{15} + X^{13} + X^{11} + X^{10} + X^9 + X^8 + X^5 + X^3 + 1$

5. Exemples de calcul de checksum

5.1. Empreinte sur 2 octets

L'exemple ci-dessous utilise la technique du modulo sur un octet (soit 255).



48 + 65 + 6C + 6C + 6F = F4 mod FF
48x1 + 65x2 + 6Cx3 + 6Cx4 + 6Fx5 = 31 mod FF

48 + 65 + 6C + 6C + 61 = F4 mod E6
48x1 + 65x2 + 6Cx3 + 6Cx4 + 61x5 = EB mod FF

Le premier octet de l'empreinte ne détecte pas une interversion de deux caractères.

Le deuxième octet réduit la probabilité de ne pas détecter d'erreur de transmission.

7 High-level Data Link Control : protocole de niveau 2 du Modèle ISO/OSI, dérivé de SDLC (Synchronous Data Link Control)

5.2. Hachage

Exemple de hachage par décalage bit à bit.

```
/************************************************************
*                                                           *
* Fonction de hashage adapté de "Compilers : Principles,    *
* Techniques, and Tools, de Alfred V. AHO, Ravi SETHI,      *
* et Jeffrey D. ULLMAN, qui l'attribuent à P. J. WEINBERGER *
*                                                           *
************************************************************/

const int myhash(const void *cle, const int PRIME_TBLSIZ)
// PRIME_TBLSIZ est la taille réelle de la table
{
  const char      *ptr = cle;
  int             val = 0;

  // Hache la clé à l'aide d'opérations bit à bit
  while ( *ptr ) {
      val = (val << 4) + (*ptr++);

      int tmp;
      if ( tmp = (val & 0xf0000000) ) {
            val = val ^ (tmp >> 24);
            val = val ^ tmp;
            }
      }

  // empêche le retour de toute valeur négative
  return (val % PRIME_TBLSIZ) < 0
      ? 0 - (val % PRIME_TBLSIZ)
      : val % PRIME_TBLSIZ ;
}
```

5.3. Complément à 255

Le calcul de checksum utilisé ci-dessous est employé dans le protocole ZigBee.

On garde le dernier octet de la somme ASCII des caractères envoyés avec son complément à 255.

```
const int setCheckSum(const void *frame)
{
  const char  *ptr = frame;
  int  sum = 0;

  while ( *ptr )
    sum += *ptr++;

  return 0xFF - (sum & 0xFF);
}
```