Insper

Camada Física - Códigos detectores de erro

Rafael Corsi - rafael.corsi@insper.edu.br

Agosto - 2017

Códigos detectores de erro

Em uma comunicação, não importa o meio pela qual os dados estão sendo enviados, não se pode garantir que o dado que sai de um nó chega integro ao outro ponto. Devemos implementar formas de detecção e correção dessas mudanças. Podemos diversos mecanismos de verificação da integridade dos dados, a complexidade desses mecanismos aumentam conforme subimos nas camadas do protocolo.

Comunicação entre dispositivos normalmente são baseados na transmissão de zeros e uns (sinais binários) o que limita muito as formas de sinalização (pense em uma conversa, se alguém falar com você de forma embaralhada facilmente você percebe que não entende a conversa).

Camada física : bit de paridade

Um mecanismo muito utilizada na camada física para a transmissão de bytes entre um ponto e outro é o do bit de paridade. O bit de paridade é um simples mecanismo de detecção de erros na transmissão onde adiciona-se um bit a palavra (BYTE) que será transmitida para forçar a soma dos bits serem **Ímpar** ou **Par** (paridade ímpar ou paridade par).

Imagine por exemplo que a interface física enviará 0xAE entre dois pontos, a representação em binário (dado serializado) será :

```
10101110
```

Se realizarmos a soma dos bits 1, obtemos um valor par (6). O que o bit de paridade (P) faz é acrescentar 1 ou 0 a palavra de 8 bits forçando-a para Par ou Ímpar, dependendo da definição do protocolo.

```
10101110 P
```

Se um protocolo estabelece paridade ímpar como padrão, a soma dos bits da mensagem anterior deveria ser ímpar e não par, para isso utiliza-se o bit de paridade para tornar a soma ímpar :

```
10101110 1
```

Nesse caso a soma dos bits 1 é igual a 7, ou seja, ímpar.

Agora imagine que aconteça um erro na transmissão onde um dos bits mude de 0 para 1:

```
11101110 1
 ^
 o bit_6 mudou durante a transmissão

soma = 8, mas deveria ser Ímpar. Dado com defeito !
```

Questão :
Escreva o dado em binário e adicione um bit de paridade **PAR** para os seguintes casos :
a. 0xFF, b. 0xAC c. 0x04 d. 0x21

Questão : Os seguintes bytes (sem paridade) foram recebidos, detecte qual apresenta erro.

Palavra	Erro ?
00100110	
10100111	
01100101	
10111111	

Questão : Os seguintes bytes (com paridade ímpar) foram recebidos, detecte qual apresenta erro.

Palavra	Erro ?
001001101	
101001110	
011001010	
101111111	

> Questão
> O que aconteceria se tivermos uma inversão de dois bits durante a transmissão ? O bit de paridade é capaz de detectar isso ? Explique.

> Questão
> Qual o overhead de adicionarmos um bit de paridade a cada 8 bits ?

> Questão
> Existe vantagem em usar paridade ímpar / paridade par ?

> Questão
> Conseguimos com a paridade corrigir o dado que chegou errado ?

O bit de paridade é bastante utilizado em comunicações em geral pois adiciona pouco overhead ao dado, porém apresenta problemas com a detecção de erros quando acontece a inversão simultânea de mais um bit durante a transmissão. Outros métodos surgiram para contornar esse problema.

Camada de enlace : CheckSum (CS)

A evolução natural do bit de paridade envolve a verificação não mais individual de um byte mais sim a verificação de um conjunto de bytes (pode ser por exemplo o conteúdo do payload ou HEAD). Similar ao bit de paridade, faz-se a soma dos valores de todos os bytes do conjunto e transmite-se essa soma junto com os dados:

```
| Byte 0 | Byte 1 | Byte 2 | ... | Byte n-1 | CheckSum |
```

Por exemplo :

```
Byte 0  :        10110101
Byte 1  :        11000000
Byte 2  :        00001101
Byte 3  :      + 11100011
                 ----------
CheckSum:        01100101
```

> Questão :
> Calcule o CheckSum (de 8 bits) dos dados : 0xFA 0x00 0xC1 0x11 0x02

> Questão :
> Escreva uma função em python que calcule o CheckSum (de 8 bits) de um conjunto de bytes (bytearray).

Por causa dessa matemática, se ocorrer inversão de bits em qualquer uma das "colunas" (bit 7..0) o valor do checksum transmitido será diferente do calculado e o receptor saberá que os dados não estão corretos.

Exemplo : se durante a transmissão dos 4 bytes definidos anteriormente houver uma mudanças em alguns bits (inclusive do checksum) o valor do checksum calculado será diferente do recebido, indicando erro na recepção:

```
             Byte 0   Byte 1   Byte 2   Byte 3     CS
Original : 10110101 11000000 00001101 11100011 01100101
-----------------------------------------------------------
Recebido : 10110101 11001000 00001101 11100111 11100101
                         ^                   ^   ^        :erros

     10110101
     11001000
     00001101
   + 11100111
     ---------
     01110001   : CheckSum Calculado

     01110001 != 11100101 : Difere do CheckSum recebido
```

> Questão :
> Quais a diferença entre CheckSum de 8 bits e de 16 bits ?

O CheckSum possui um defeito: caso o erro aconteça na mesma coluna (invertendo dois bits do mesmo peso), o checksum não será alterado e **o erro não será detectado** :

```
             Byte 0   Byte 1   Byte 2   Byte 3     CS
Original : 10110101 11000000 00001101 11100011 01100101
-----------------------------------------------------------
Recebido : 00110101 11000000 10001101 11100011 11100101
           ^                 ^                           :erros
```

```
Byte 0  :        00110101
                 ^
Byte 1  :        11000000
Byte 2  :        10001101
                 ^
Byte 3  :      + 01100011
                ----------
CheckSum:        01100101:
```

```
01110101 = 01100101 : CheckSum calculado = CheckSum recebido
```

Porém houve erro na transmissão/recepção dos dados. O CheckSum falhou.

CRC

Checksums são algorítimos simples e que podem ser facilmente implementados em hardware (lembra de elementos ?) porém comunicações mais seguras não podem ...

O *cyclic redundancy check* (CRC) é uma evolução do checksum porém utiliza divisão no lugar de soma. A capacidade do CRC de detecção de erros é muito mais forte que a do checksum. Este código se baseia em divisão de polinômios :

- Interpretar os dados como sendo coeficientes de um polinômio
- Cada código CRC tem um polinômio especial a ser usado como divisor
- O resto da divisão forma os bits de redundância do código

O CRC tem a capacidade de conseguir além de detectar o erro, de corrigir em algumas situações o dado corrompido.

Polinômio

O CRC utiliza um polinômio para realizar a divisão, o polinômio define os coeficientes de :

```
x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x^1 + x^0
```

Por exemplo: O polinômio **CR-4 : x^4+x+1**, define a seguinte palavra:

```
x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x^1 + x^0
 0     0     0     1     0     0     1     1
```

Palavra binária resultante do polinômio : 00010011

Os polinômios são as chaves para o CRC, a escolha do polinômio define o quão bom o seu tratamento de erro será [4].

> Questão:
> Ache a palavra binária para os polinômios a seguir :
> CRC5 = xˆ5 + Xˆ2 + 1 CRC6 = xˆ6 + x + 1 CRC16 = x^{16+x}15+xˆ2+1

> Questão:
> Dado a palavra binária a seguir, encontre seu polinômio :
> CRC10 : 011000110011 CRC15 : 1100010110011001

Algorítimo

Pega-se a palavra que deseja calcular o CRC e faz-se a divisão binária, aplicando um XOR a cada termo da divisão, o resto da divisão é o valor do CRC que deve ser anexada a mensagem para correção de erros. Conforme exemplo a seguir onde deseja-se calcular o CRC para a palavra 101101010000 com o polinômio x^{4+x}1+1:

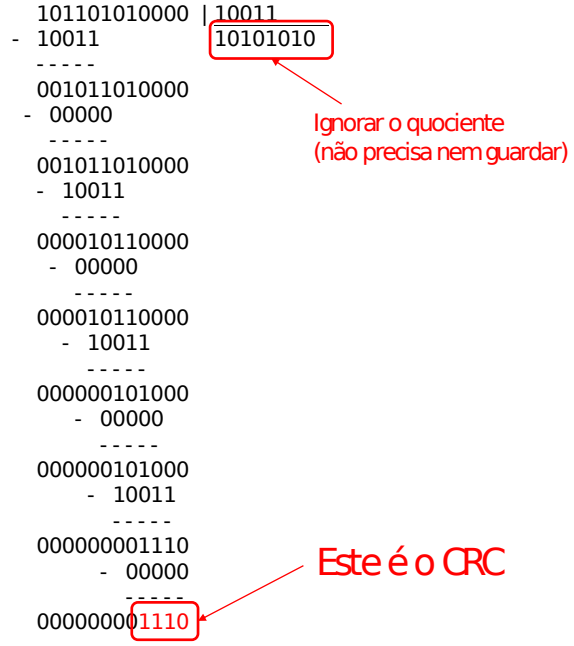

Figura 1: Cálculo CRC

Referências :

- [1] : https://technet.microsoft.com/en-us/library/cc757419(v=ws.10).aspx

- [2] : https://mrncciew.com/2014/11/03/cwap-802-11-fragmentation/
- [3] : https://barrgroup.com/Embedded-Systems/How-To/Additive-Checksums
- [4] : https://barrgroup.com/Embedded-Systems/How-To/CRC-Math-Theory