Desarrollo e implementación de códigos polinomiales para la detección de errores en sistemas digitales (CRC_k).

Sebastián Valencia Calderón
Juan Camilo Bages
Universidad de los Andes

Abril 11, 2014

El siguiente informe, documenta el proceso de análisis y desarrollo de algoritmos de detección de errores en la transmisión para datos en redes digitales. Sin embargo, la implementación no se desarrolla en el ámbito de las redes de comunicación, sino, más bien, en un marco general para entender el funcionamiento y utilidad de los algoritmos de detección de errores en problemas prácticos de ingeniería. El proceso de análisis, se realizará usando pseudocódigo (omitiendo detalles de especificación y análisis de eficiencia), mientras la implementación y desarrollo práctico se realizarán ambos usando el lenguaje de programación C. En concreto, el algoritmo que se analizará, diseñará, e implementará, será CRC (Cyclic redundancy check, por sus siglas en inglés), por sus facilidades de implementación y desarrollo. La técnica original, fue planteada y diseñada por Richard Hamming. Está técnica, es usada actualmente por la IEEE, para internet.

1 Introducción

En transmisión digital de datos, los erores ocurren cuado un bit es alterado entre el proceso de transmisión y recepción; es decir, *0b1* es transmitido mientras se recibe *0b0*, o viceversa. Además de este tipo de errores, la perdida de información puede ocurrir, lo que acarrea problemas a la hora de la reconstrucción de la información, y de la comprensión de la misma. Los errores pueden ser más graves que la pérdida o alteración de un sólo bit, pérdidas o alteraciones más grandes pueden ocurrir, es decir, una trama entera de datos, puede verse afectada, o puede sufrir alteraciones importantes para la integridad de los datos.

La presencia de errores, es independiente al diseño del sistema de transmisión, o del diseño de complejos protocolos de comunicación. Por lo tanto, es importante, preveer la presencia de errores y tratar de detectarlos y en ocasiones corregirlos. Para este propósito, existen, ciertas técnicas de deteciión y corrección de errores sobre tramas

de datos transmitidas a través de una red. Por pragmatismo, y eficiencia, se omiten especificaciones de diseño y procesos de los algoritmos diseñados en el largo desarrollo de las redes digitales, para compensar esto, la bibliografía ofrece teoría sobre el plantemiento teórico de estos procesos algoritmicos.

2 Marco teórico

Una técnica común pero de todas maneras poderosa, de detección de errores es CRC (debe ser claro, que el objetivo general en el diseño de algoritmos de detección de errores, es maximizar la probabilidad de detectar errores usando únicamente un número mínimo de bits redundantes), el cual se basa sobre un área poderosa de las matemáticas para lograr éste objetivo. Los fundamentos teóricos de ésta técnica es la teoría de campos algebráicos. CRC, a veces se refiere como una técnica de códigos polinomiales, debido a la posibilidad de representación de tramas de bits usando polinomios. En general, un mensaje de $(n+1)$-bit, puede representarse usando un polinomio de grado n, usando como coeficiente de cáda término, el bit correspondiente por posición en el mensaje total. Por ejemplo, para el mensaje de 8-bit *0b10011010*, existe un polinomio representativo $M(x)$ donde $M(x) = 1\times x^7+0\times x^6+0\times x^5+1\times x^4+1\times x^3+0\times x^2+1\times x^1+0\times x^0 = x^7+x^4+x^3+x^1$. Por lo tanto, el proceso de transmisión puede entenderse como un proceso de intercambio de polinomios.

Hasta ahora, nada se ha dicho sobre la naturaleza computacional del CRC, sino, se han explicado las bondades de la corrección y detección de errores en datos. Antes de proceder con una definición y descripción meramente formal de la derivación de los códigos de redundancia cíclica, se da una descripción informal y general de la naturaleza del algoritmo. Un matemático, entiende CRC como la computación del residuo de la división de dos polinomios con coeficientes en $\mathbb{Z}_2$. Un científico de la computación, comprende CRC como la computación del residuo de la división de dos números binarios, uno representando el mensaje, y otro, un divisor fijo. Para los criptógrafos, CRC, es la computación de una operación matemática sobre el campo de Galois de orden 2 ($\mathbb{GF}_2$). Para los programadores, CRC, es el proceso que itera sobre un mensaje y usa una tabla para obtener valores aditivos para cada paso de la iteración. Para los ingenieros y arquitectos de Hardware, CRC, es el resultado computacional del proceso de un circuito lógico que usa división y repetición.

La descripción anterior, sugiere que CRC, es una técnica basada en poderosas matemáticas, es útil para cualquier longitud de mensaje, es un proceso compacto, con una fácil implementación en hardware, y por lo tanto en software (siempre y cuando se use un lenguaje con facilidades de manejo de bajo nivel.

Para formalizar el proceso, considérese que el transmisor y receptor, deben que ponerse de acuerdo en la elección de un polinomio divisor $C(x) \in \mathbb{P}_k$. Cuado el transmisor desea enviar o transmitir un mensaje $M(x)$ de $n+1$ bits, lo que se envía realmente es el mensaje de $(n+1)$-bit mas k bits de redundancia para la posterior verificación del men-

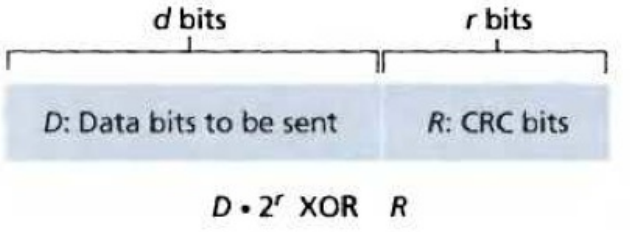

Figure 1: Mensaje enviado con la trama de redundancia

saje por parte del receptor. El mensaje completo, incluyendo los bits de reduncdancia es $P(x)$. Las matemáticas básicas, ayudan a inferir que $P(x)$ es divisible por $C(x)$. Si $P(x)$ es transmitido a través de un canal y no hay errores en la transmisión, el receptor al dividir $P(x)$ por $C(x)$, debe obtener cero. De lo contrario, sabemos que ha ocurrido un error. Para entender mejor el proceso, es mejor entender un poco de aritmética binaria, lo cual facilitará la futura implementación en C. La figura 1, muestra el mensaje transmitido con la redundancia incluida (tomado de Kurose & Ross, ver bibliografiía).

Al lidiar con números en $\mathbb{Z}_2$, la aritmética, se simplifica al usar operaciones modulo 2. Para resumir los vericuetos teóricos, se resumen las propiedades de está aritmética a través de polinomios. Para considerar esto, considerese una función $G : \mathbb{P}_k \to \mathbb{Z}$, cuya descripción o especificación es tomar un polinomio y devolver su grado, o mayor potencia de la variable de evaluación.

- $B(x) \mid C(x) \iff G(B) \geqslant G(C)$
- $B(x) \bmod C(x) = B(x) \oplus C(x)$

Una vez dispuestas estas reglas, es posible realizar división de tramas de bits. Si se quiere transmitir o crear un polinomio para la transmisión derivado del mensaje original $M(x)$, que sea k bits más largo que $M(x)$, y sea divisible por $C(x)$, se puede realizar el siguiente proceso:

- $M(x) \times x^k$, es agregar k ceros al final del mensaje
- $R(x) = T(x) \bmod C(x)$
- $T(x) - R(x)$

Al final es fácil ver que el mensaje resultante es $M(x)$, seguido por el residuo obtenido en el paso 2. Como ejemplo, consideremos el mensaje 10011010, y el código es $CRC_3 = 1101$. Primero, multiplicamos el polinomio característico del CRC, obtenemos 10011010000. Ahora se divide esto por $C(x) = 1101$. El proceso de muestra en la figura 2 (tomado de Peterson & Davie). Nótese que el proceso es la utilización consecutiva de la operación

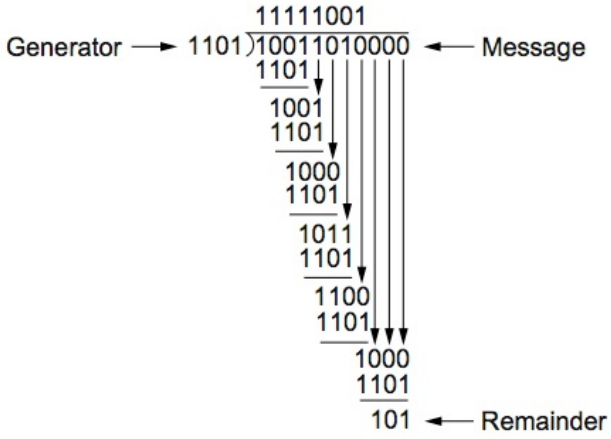

Figure 2: Calculo de CRC, usando división polinomial

XOR. Finalmente, el mensaje transmitido es 10011010101, donde los tres últimos dígitos son correspondientes al residuo.

A continuación se muestra una table con los polinomios más comunes en la práctica.

CRC más usados en la práctica	
CRC_8	$x^8 + x^2 + x^1 + 1$
CRC_{10}	$x^10 + x^9 + x^5 + x^4 + x^1 + 1$
CRC_{12}	$x^12 + x^11 + x^3 + x^2 + x + 1$
CRC_{16}	$x^16 + x^15 + x^2 + 1$
$CRC_{16'}$	$x^16 + x^12 + x^5 + 1$
CRC_{32}	$x^32 + x^26 + x^23 + x^22 + x^16 + x^12 + x^11 + x^10 + x^8 + x^7 + x^5 + x^5 + x^2 + x + 1$

3 Desarrollo

Para asistir el proceso de depuración, se presenta un procedimiento para imprimir una entidad sin signo (unsigned), dado el valor, y la cantidad de bits usabas para la representación. Es decir, si se llama binaryRepresentation(36, 16), esto debe imprimir 0000000000100100. En la siguiente página se muestra el código. Para el desarrollo de la función $darTamanho() : \mathbb{U} \rightarrow \mathbb{Z}$, se razonó de la siguiente manera. La especificación del procedimiento es calcular el número de bits necesarios para representar un número decimal en el sistema binario.

Teorema: El número de bits necesarios para representar $n \in \mathbb{Z}$ en $\mathbb{Z}_2$ está dado por $\lfloor \log_2 n \rfloor + 1$.

Demostración : Sea m el número de bits necesarios para rpresentar $n \in \mathbb{Z}$, esto es:

$$n = b_{m-1} \times 2^{m-1} + b_{m-2} \times 2^{m-2} + \ldots + b_1 \times 2 + b_0$$

$$n = \sum_{i=0}^{m-1} b_i \times 2^i \mid b_{m-1} = 1 \wedge b_i \in \mathbb{Z}_2 \wedge i \in [0, m-1)$$

$$n \leqslant \sum_{i=0}^{m-1} 2^i = 2^m - 1 < 2^m$$

$$\log_2 n < m$$

Dado que $b_{m-1} = 1$, tenemos una cota inferior para n, es decir, $n \geqslant 2^{m-1}$ y por lo tanto, $\log_2 n \geqslant m - 1$, dado que $m \in \mathbb{Z}$, $m - 1 = \lfloor \log_2 n \rfloor$, despejando tenemos que, $m = \lfloor \log_2 n \rfloor + 1$. Esto se considera una prueba formal de que darTamanho(unsigned short arg), puede escribirse como $floor(log2(arg)) + 1$.

```
void binaryRepresentation(unsigned number, int bits) {
  int need = darTamanho(number), counter;
  char array[bits];
  for(counter = 0; counter < bits; counter++)
    array[counter] = '0';
  counter = bits - 1;
  while(number != 0) {
    int flag = number % 2;
    array[counter] = (flag == 1)? '1' : '0';
    counter--;
    number = number / 2;
  }
  for(counter = 0; counter < bits; counter++)
    printf("%c", array[counter]);
  printf("\n");
}
```

Para el desarrolo de la funcción “unsigned short leerPrimerBloque(TEXTO *datos)”, esta se especifica como la que extrae los dos primeros bytes para conformar un short, para esto, se ingresa por valor al mensaje de la estructura, a su tamaño, y se corre un número binario, las posiciones necesarias para llenar un short. El proceso está explícito en el código. Para la función “void corregirPrimerBloque(unsigned short * primerBloque)”, se extraen los bits desde la posición más significativa hasta completar el tamaño de k, para esto, considérese el siguiente procedimiento, cuyo objetivo es extraer los n- bit de x desde p hacia atrás, asumiendo que la posición menos significativa es la 0, y que $p, n \in \mathbb{Z}$. Entonces, $getbits(x, 4, 3)$, retorna los bits en la posición 4, 3 y 2.:

```
unsigned getbits(unsigned x, int p, int n) {
  return (x >> (p+1-n)) & ~(~0 << n);
}
```

En el anterior procedimeinto, la expresión $(x >> (p+1-n))$, mueve el campo deseado a la derecha de la palabra. 0 es $1111 \ldots 1111$, mover a la izquierda n bit con la expresión $(0 << n)$, acomoda ceros en las n posiciones menos significativas. Complementando con , permite devolver los n bit más a la derecha. Siuiendo este razonamiento, puede decirse que dado el primer bloque, es necesario obtener desde la posición más significativa hasta k del primer bloque.

Para reemplazar ceros, descrita a continuación, se toma unos Bytes de solo unos y se corre $k-1$, tal y como se desea, se toma en AND con el residuo, y esto se guarda en la variable shift, luego, se cambia el valor del mensaje por su valor actual OR lo que se había calculado antes.

```
void reemplazarCeros(TEXTO * txt, unsigned short residuo) {
  unsigned short shift = residuo & ~(~0 << k - 1);
  *txt->mensaje = *txt->mensaje | shift;
}
```

Para agregar bit, la función que agrega el bit que llega por parámetro a txt, se debe entender el bit que llega, es decir pasar de char a int, y luego se agrega por valor. Para esto, se corre *txt al lado del bit más significativo y se hace OR con el bit que llega, pues si es cero, para el computador es $0000 \ldots 0000$, si es uno, para el computador es $000 \ldots 0001$.

```
void agregarBit(unsigned short * txt, unsigned char bit) {
  int realBit = (bit == '0')? 0 : 1;
  *txt = (*txt << 1) | realBit;
}
```

El razonamiento para “unsigned char bajarDigito(unsigned char * msj, int bitpos)”, es similar, pues se debe reconocer el bit, que representa la posición, y posteriormente devolver el bit en esta posición. Lo que debe hacer esta función es devolver el dígito en la posición que llega por parámetro. Para su desarrollo, se usó el razonamiento expueso en getbit.

```
unsigned char bajarDigito(unsigned char * msj, int bitpos) {
  // (*msj >> bitpos) & 1
  int flag = (*msj >> bitpos) & ~(~0 << 1);
  return (flag == 0) ? '0' : '1';
}
```

La estructura de las funciones hasta ahora definidas, se observan en el siguiente código, los detalles, están el el código real.

```
#include <stdio.h>
#include <stdio.h>
#include <stdbool.h>
#include <math.h>
#include <assert.h>

int k;

typedef struct text {
  unsigned char *mensaje;
  int tamanho;
} TEXTO;

int darTamanho(unsigned short bloq) {
  return floor(log2(bloq)) + 1;
}

void corregirPrimerBloque(unsigned short * primerBloque) {
  int tamanioMax = darTamanho(*primerBloque);
  int tamanioMin = k;
  int p = tamanioMax - 1;
  int n = tamanioMin;
  *primerBloque = (*primerBloque >> (p + 1 - n) & ~(~0 << n));
}

void agregarBit(unsigned short * txt, unsigned char bit) {
  int realBit = (bit == '0')? 0 : 1;
  *txt = (*txt << 1) | realBit;
}

void reemplazarCeros(TEXTO * txt, unsigned short residuo) {
  unsigned short val = residuo & ~(~0 << (k - 1));
  * txt -> mensaje = * txt -> mensaje | val;
}

unsigned short leerPrimerBloque(TEXTO *datos) {
  int sizeOfMensaje = darTamanho(*datos->mensaje);
  unsigned short val = *datos->mensaje & (~0 << (sizeOfMensaje - 16)
   );
  return val;
}

unsigned char bajarDigito(unsigned char * msj, int bitpos) {
  int flag = (*msj >> bitpos) & ~(~0 << 1);
  return (flag == 0)? '0' : '1';
}

void binaryRepresentation(unsigned number, int bits) {
  int need = darTamanho(number);
  assert(need <= bits);
```

```
  int counter;
  char array[bits];
  for(counter = 0; counter < bits; counter++)
    array[counter] = '0';
  counter = bits - 1;
  while(number != 0) {
    int flag = number % 2;
    array[counter] = (flag == 1)? '1' : '0';
    counter--;
    number = number / 2;
  }
  for(counter = 0; counter < bits; counter++)
    printf("%c", array[counter]);
  printf("\n");
}

int main(int argc, char const *argv[]) {
  int machine = 16;
  int generador = 350;
  k = darTamanho(generador);
  printf("%d\n", darTamanho(350));
  return 0;
}
```

Finalmente, para la división, se realiza el proceso tal y como se explica en la guía, es decir, primero, se obtiene el bit en la posición actual usando las funciones previamente definidas, posteriormente, se agrega al final del residuo, y se repite hasta alcanzar el tamao de k, en cuyo caso, se retorna agregan los bits del residuo en la cola del mensaje.

```
unsigned short division(TEXTO * txt, unsigned short generador)
{
  unsigned short primerBloque=leerPrimerBloque(txt);
  int tamanhoResiduo = darTamanho(primerBloque);
  int posActual = 16;
  unsigned short residuo;
  unsigned char bitBajar;
  int fin = ((txt->tamanho - 2) * 8) + (k - 1);
  if (k<tamanhoResiduo) {
    corregirPrimerBloque(&primerBloque, k);
    posActual = 16-(tamanhoResiduo - k);
  }

  residuo = primerBloque;
  while (posActual < fin)
  {
    if (tamanhoResiduo < k)
    {
      unsigned char bitPosActual = bajarDigito((txt->mensaje),
   posActual);
      agregarBit(&residuo, bitPosActual);
```

```
        posActual++;
22    }
      tamanhoResiduo = darTamanho(residuo);
24    if(tamanhoResiduo == k)
        reemplazarCeros(txt, residuo);
26  }
  return residuo;
28}
```

References

[1] Larry L. Peterson, *Computer Networks: A Systems Approach*. Elsevier Computers, Massachusetts, 5th Edition, 2011.

[2] James F. Kurose, *Computer Networking: A Top-down Approach*. Pearson, 6th Edition, 2013.

[3] William Stallings, *Data and Computer Communications*. Pearson Education, 8th Edition, 2007.

[4] Behrouz A. Forouzan, *Data Communications and Networking*. Huga Media, 4th Edition, 2007.

[5] Douglas Comer, *Computer Networks and Internets: With Internet Applications*. Pearson/Prentice Hall, 5th Edition, 2004.

[6] Andrew S. Tanenbaum,, *Computer Networks: Pearson New International Edition*. Pearson Education, Limited, 1st Edition, 2013.

[7] Peter J. Cameron,, *Introduction to Algebra*. Oxford University press, 2nd Edition, 2008.

[8] David S. Dummit, *Abstract Algebra*. John Wiley and Sons. Inc, 3rd Edition, 2004.

[9] Joseph A. Gallian, *Contemporary abstract algebra*. Cengage learning, 7th Edition, 2010.

[10] John G. Proakis, *Communication systems engineering*. Prentice Hall, 2nd Edition, 2002.

[11] Gilles Brassard,, *Algorithmics: theory and practice*. Prentice Hall,, 1st Edition, 1988.

[12] Brian W. Kernighan, Dennis Ritchie *The C Programming Language*. Pearson Education, 2nd Edition, 1988.

[13] K. N King *C Programming: A Modern Approach*. Granite Hill Publishers, 2nd Edition, 2010.