CRC(Cyclic Redundancy Check) 검사

1957년부터 연구되어 온 순회부호(cyclic code)는 풍부한 대수적 구조 및 단순한 하드웨어 구성 으로 현재 가장 많이 응용되고 있다. 본 고에서는 순회부호의 원리를 응용한 에러 검출 알고리즘인 CRC에 대하여 CRC-32 부호를 중심으로 에러검출 성능을 분석하였다.

에러제어 코딩에 대한 연구는 1948년 C.E. Shannon이 발표한 <통신에 관한 수학적 이론(A Mathematical Theory of Communication)>이라는 논문부터 시작되었다. Shannon은 이 논문에서 정 보에 확률 개념을 도입하여 정보를 비트로 표현하고 잡음 채널과 무잡음 채널에서 에러 없이 정보 를 전송하기 위한 이론이 제창하였다. 그 이후 잡음이 있는 환경에서 에러 제어를 위한 인코딩과 디 코딩에 관한 연구가 활발히 수행되고 있으며, 에러 제어를 위한 부호(code)의 사용은 통신 시스템과 디지털 컴퓨터 설계에 필수적인 요소가 되었다.

에러 제어를 위한 부호에는 블록부호(block code)와 길쌈부호(convolutional code)로 나눌 수 있 다. 블록부호는 k비트의 정보를 n비트의 부호워드로 만들어 내는 것으로 선형부호(linear code)와 순 회부호(cyclic code)로 나눌 수 있다. 반면, 길쌈부호는 출력 시퀀스가 현재의 입력 뿐만 아니라 과거 의 입력 시퀀스에 의해서도 영향을 받는 부호를 말한다. 순회부호는 E. Prange에 의해 1957년부터 1957년에 걸쳐 발간된 일련의 기술보고서들에서 논의되기 시작하여, BCH code, Reed-Solomon code 로 발전하였다. 이 순회부호는 풍부한 대수적 구조를 갖고 있어 많은 연구 성과가 발표되었고, 고속 의 쉬프트 레지스터(shift register)를 기반으로 인코더와 디코더를 간단히 구현 할 수 있으므로 CD player 및 giga bit/sec 급 고속 통신 등 다양한 분야에서 널리 실용화 되어 있다.

CRC(Cyclic Redundancy Check)는 순회부호의 일종으로 Shortened Cyclic code라고도 하며, 인 코더 및 디코더의 구현히 대단히 간단하고, 에러 검출을 위한 오버헤드가 적고, 버스트 에러를 포함 한 에러검출에 매우 좋은 성능을 갖는 것을 특징으로 한다.

CRC의 원리

CRC의 원리는 나눗셈을 한 결과의 나머지를 보낼 데이터의 잉여분으로 덧붙여 보내서 수신단 에서 이 잉여분과 함께온 데이터를 원래의 데이터를 나눈값으로 나누었을 때 나머지가 0인지 검사 해서 오류를 검출하는 코드이다. 여기서 나머지는 CRC에서 가장 중요한 부분으로 Frame Check Sequence(FCS)라고 부른다. 데이터 전송시에 송신측에서는 CRC를 검사하기 위해서 각 데이터 프레임 마다, 전송된 프레임의 에러를 검출할 수 있는 여분의 정보(FCS)를 포함해서 전송한다.

이러한 CRC는 이진수를 기본으로해서 모든 연산이 동작한다. 즉, 모든 스트림 데이터를 바이너 리 폴리노미얼처럼 다룬다. 원래의 데이터 프래임이 주어지면, 전송하는 측에서는 원래의 데이터 프 래임을 위한 FCS를 생성한다. FCS를 생성하기 위해서는 나눗셈을 위한 젯수인 CRC 폴리노미얼이 필요하다. 앞에서 이야기했듯이 FCS를 생성하는 방법은 전송할 데이터 프래임을 CRC 폴리노미얼로 나누면, 나머지 값이 나오게 되는데, 이 값이 Frame check Sequence(FCS)가 된다. FCS는 결과 프 래임(전송될 프래임 : 원래의 데이터 프래임과 FCS가 연결된 프래임)이 미리 정의된 폴리노미얼에 의하여 정확하게 나누어질 떨어질 수 있도록 원래의 데이터 프래임의 뒤에 붙여서 송신하게 된다. 여기서, 미리 정의된 폴리노미얼을 devisor 또는 CRC Polynomial이라고 한다. 결과 프래임을 수신단 에서 받은 후에 CRC 검사를 하게 되는데, 검사하는 방법은 수신된 데이터 프래임을 전송시에 사용 한 것과 같은 CRC 폴리노미얼로 나누어서 나머지를 검사한다. 이때 나머지가 0이 아닌 수가 되면, 전송시에 에러가 발생한 것이다. 따라서 에러에 대한 적당한 동작을 취해주면 된다.

아래에서 CRC의 절차를 소개하는데 필요한 몇가지 정의된 문자를 설명한다.

M – FCS가 더해지기 전의 전송될 원래의 데이터 프레임, k bits.

F – M에 더해질 FCS 프레임, n bits.

CRC 원리의 요점은 T/P의 나머지가 0이 되도록 FCS를 생성하는 것이다. 따라서 다음의 과정 을 거쳐서 CRC를 검사할 수 있다.

① $T = M*x^n+F$

이식은 T가 M과 F를 연결한 것이므로, M을 n 비트 만큼 좌로 쉬프트를 한 후에, 하위 비트들 에 F를 더해서 T를 생성하는 것이다. 여기서 전송하기를 원하는 프레임 T(CRC Polynomial로 나누 어 나머지가 0이 되도록 만들어진 프레임)를 구하기 위해서는, 원래의 데이터 프레임인 M에 대한 적당한 FCS 프레임 F를 구해야 한다.

② $M*x^n$을 P로 나누면, 몫과 나머지가 나온다.

$M*x^n / P = Q + R/P$

여기서 식①에서의 F를 R이라 하면,

③$T = M*x^n + R$이 된다.

④ T가 CRC 폴리노미얼로 나누어 떨어지면,

$T / P = (M*x^n / P) + R/P$

$= (Q + R/P) + R/P = Q + (R + R)/P$

이진연산으로 XOR을 하게되면, 어떤 숫자라도 0이된다.

⑤ 따라서, $T/P = Q$ 가 된다.

위의 과정을 간단하게 정리하면, 전송측에서는,

1. 원래의 프레임을 얻는다.
2. 원래의 프레임을 n 비트 쉬프트 하고, P 로 나눈다.
3. 나머지를 통해서, FCS를 얻는다.
4. 원래의 프레임에 FCS를 덧붙이고, 결과 프레임을 전송한다.

수신측에서 CRC 에러 검출은 다음과 같은 과정을 거친다.

1. 프레임을 받는다.
2. P로 받은 프레임을 나눈다.
3. 나머지를 체크한다. (0이 아니면, 전송시 에러 발생)

아래에서 CRC 생성 및 에러 검출 과정의 예를 보이겠다.

M = 10001(5bits)
P = 1101(4bits) ⇒ x4+x3+x2+1 (CRC-4)
F = ????
F = R = 0101

CRC의 응용

IBM 기종의 플로피 디스크에서는 16 비트 CRC를 사용하며 아주 가끔 CRC 에러를 내기도 한 다. 또 흔히 사용하는 압축 프로그램들이 데이터 복구 과정에서 에러를 체크하는 방식으로 사용한다. 참고로, LHA는 16 비트 CRC를 사용하며 IBM 플로피 디스크 16 비트 CRC 반전시켜서 사용하고 PKZIP과 ARJ는 32 비트 CRC를 사용한다.

이 CRC 를 응용한다면 첫째로 바이러스 침투 여부를 검사하는 프로그램이다. 32비트 CRC를 속 이고 침투할 수 있는 바이러스는 존재하지 않는다. 또 데이터의 암호화와 데이터의 변화 여부를 1 패스에 구현할 수 있는 곳에도 응용이 된다. 해커 침입 방지 프로그램같은 것에도 응용 가능하다.

많이 사용되는 CRC 폴리노미얼

CRC-12 : x12 + x11 + x3 + x2 + x + 1
CRC-16 : x16 + x15 + x2 + 1
CRC-ITU-T : x16 + x12 + x5 + 1
CRC-32 : x32 + x26 + x23 + x22 + x16 + x12 + x11 + x10 + x8 + x7 + x5 + x4 + x2 + x + 1

CRC 소스

```
/*  CRC-32b version 1.03 by Craig Bruce, 27-Jan-94
**
**  Based on "File Verification Using CRC" by Mark R. Nelson in Dr. Dobb's
**  Journal, May 1992, pp. 64-67.  This program DOES generate the same CRC
**  values as ZMODEM and PKZIP
**
**
*/
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <time.h>
#include <math.h>
#define OCTET               256
#define ECTET_MAX           255
#define FRAME_SIZE          1024
#define FCS_SIZE            4
#define TOTAL_SIZE          1028
#define ERROR_STEP          200
#define ERROR_MAX           4100
#define TEST_TIMES          1000

void frame_gen(char *frame) ;
void error_gen(char *frame, int err_cnt) ;
void crcgen(void) ;
void err_pos_gen(int err_cnt) ;
int get_fcs(char *frame, int frame_size) ;

unsigned long    crcTable[256];
int              error_pos[TOTAL_SIZE*8] ;
```

```
/*************************************************************************/
int main( int argc, char *argv[] )
{
        int             err_cnt, i, count ;
        unsigned long    crc;
        time_t          seed ;
        char             in_frame[TOTAL_SIZE], out_frame[TOTAL_SIZE] ;
        time(&seed) ;
        srand(seed) ;
        crcgen();        /* 초기화... */

        printf("   *** CRC 에러 검출 시뮬레이션 ***₩n₩n") ;
        printf("에러수₩t₩t전송수₩t₩t검출수₩n") ;
        for(err_cnt=ERROR_STEP ; err_cnt<ERROR_MAX ; err_cnt+=ERROR_STEP)
        /* 시뮬레이션을 에러가 1개일때 6개 일때 11개일때... 31개일때 */
        {
                count = 0 ;
                err_pos_gen(err_cnt) ;
                /* 에러가 발생할 자리를 미리 정한다. */
                for(i=0 ; i<TEST_TIMES ; i++)
                {
                        frame_gen(in_frame) ;
                        crc = get_fcs(in_frame, FRAME_SIZE) ;
                        memcpy(in_frame+FRAME_SIZE, &crc, sizeof(crc)) ;
                        memcpy(out_frame, in_frame, TOTAL_SIZE) ;
                        error_gen(out_frame, err_cnt) ;
                        crc = get_fcs(out_frame, TOTAL_SIZE) ;
                        if(crc != 0) count ++ ;
                }
                printf("%5d₩t₩t%5d₩t₩t%5d₩n", err_cnt, i, count) ;
        }
        return( 0 );
}

/*************************************************************************/
/*     void frame_gen(char *frame) ;                                     */
/*       쓰임새 : 난수발생으로 임의의 데이터 전송 프레임을 만든다.       */
/*       매개변수 :                                                      */
/*                frame - 난수 발생으로 생성한 데이터 프레임이 저장될 주소 */
/*       반환값 : 없음.                                                  */
/*************************************************************************/
void frame_gen(char *frame)
{
        int      i ;
        for(i=0 ; i<FRAME_SIZE ; i++)
                frame[i] = rand()%255 ;          /* 0-255 사이의 난수 발생 */
}

/*************************************************************************/
/*     void error_gen(char *frame, int err_cnt) ;                        */
/*       쓰임새 : 난수발생으로 임의의 에러를 생성한다.                   */
/*       매개변수 :                                                      */
/*                frame - 데이터 프레임이 저장되어 있는 주소             */
/*                err_cnt - 데이터 프레임에서 발생할 에러의 개수         */
/*       반환값 : 없음.                                                  */
/*************************************************************************/
void error_gen(char *frame, int err_cnt)
```

```
{
        int     i=0, l, k ;
        for(i=0 ; i<err_cnt ; i++)
        {
                l = error_pos[i]/8 ;
                k = error_pos[i]%8 ;
                frame[l] = frame[l] ^ (1<<k) ;
        }       /* 미리 결정한 에러가 발생할 자리의 비트를 반전시켜서
                에러를 발생시킨다. */
}

/*****************************************************************************/
/*      void err_pos_gen(int err_cnt) ;                                      */
/*      쓰임새 : 에러가 발생할 위치를 미리 결정해 놓는다.                    */
/*               속도문제 개선을 위해 미리 계산해 놓는다.                    */
/*      매개변수 :                                                           */
/*               err_cnt - 에러가 발생할 개수                                */
/*      반환값 : 없음.                                                       */
/*****************************************************************************/
void err_pos_gen(err_cnt)
{
        int     i, j, k ;
        for(i=0 ; i<err_cnt ; i++)
        {
                while(j!=i)
                {
                        k = rand()%(TOTAL_SIZE*8) ;
                        for(j=0 ; j<i ; j++)
                                if(error_pos[j]==k) break ;
                        error_pos[i] = k ;
                }
        }
}

/*****************************************************************************/
/*      void crcgen(void) ;                                                  */
/*      쓰임새 : 바이트(256개)에 대한 CRC-32에 대한 나머지값들을 계산과정    */
/*               에서의 속도문제 개선을 위해 미리 계산해 놓는다.             */
/*      매개변수 : 없음.                                                     */
/*      반환값 : 없음.                                                       */
/*****************************************************************************/
void crcgen(void)
{
        unsigned long   crc, poly;
        int     i, j;
        poly = 0xEDB88320L; /* zmodem과 pkzip에서 사용되는 공개 폴리노미얼 */
        for (i=0; i<256; i++) {
                crc = i;
                for (j=8; j>0; j--) {
                        if (crc&1) {
                                crc = (crc >> 1) ^ poly; /* 1이면, 뺀다. */
                        } else {
                                crc >>= 1; /* 0이면, 쉬프트만 */
                        }
                }
                crcTable[i] = crc;
        } /* 8비트(256)에 해당하는 CRC의 나머지 값을 미리 계산해 놓는다. */
```

```
        /* 최하위 비트가 그림에서의 최상위 비트가 된다. (역순으로 계산함...) */
}

/*****************************************************************************/
/*     void get_fcs(char *frame, int frame_size) ;                           */
/*       쓰임새 : 현재 프레임에 대한 FCS값(CRC 검사값)을 얻어온다.           */
/*       매개변수 :                                                          */
/*                frame - CRC를 check할 데이터 프레임.                       */
/*                frame_size - 데이터 프레임의 크기.                         */
/*       반환값 : 없음.                                                      */
/*****************************************************************************/
int get_fcs(char *frame, int frame_size)
{
        register unsigned long          crc ;
        int                             b, i ;
        crc = 0xFFFFFFFF ;
        /* 데이터 프레임 전체를 하나의 M으로 보고 FCS를 계산한다. */
        for(i=0 ; i<frame_size ; i++)
                crc = ((crc>>8) & 0x00FFFFFF) ^ crcTable[(crc^frame[i]) & 0xFF];
        /* 들어온 바이트 만큼 쉬프트 시키면서 나머지를 계산해 나간다. */
        return (crc^0xFFFFFFFF) ;
        /* 처음에 반전된 것을 원래대로 만든다. */
}
```