# Cyclic Redundancy Check (CRC)

To each message, we will append n additional bits, where n is a **constant.**

Let m be the message size.

Note that m is not constant, the technique works *for messages of any size.*

We assume we have a *constant* G, which is a string of n+1 bits.

These n+1 bits can be viewed as the coefficients of an nth degree polynomial.

E.g., if G = 1011 (n = 3), then G can be thought of as the polynomial

**1**⋅x3 + **0**⋅x2 + **1**⋅x + **1**⋅ x0

x3 + x + 1

**NOTE**: **ALL** our polynomials will only have coefficients of 0 or 1.

# Computing the Remainder

Let M be the original data message (with m bits)

M is just a string of bits, and can be viewed as a polynomial.

Note that M⋅xn = M concatenated with n zeroes.

If M = 1001001 and n = 4, then M⋅xn = 1001001**0000**

Let R be the remainder of (M⋅xn)/G.

Then, (M⋅xn - R)/G should give a remainder of zero (simple arithmetic).

# Sender/Receiver Operation

The sender computes R (i.e. the remainder of (M\*xn)/G), and attaches R at the end of M, and sends it over the channel.

I.e., the sender sends M;R, where ; denotes concatenation.

The channel transforms M;R into M’;R’ (i.e. corruption)

The receiver receives M’;R’ and performs (M’\*xn – R’)/G.

Let z be the remainder of (M’\*xn – R’)/G.

If z is zero, then it assumes the message is not corrupted   
(although it may be).

If z is not zero, then for sure the message is corrupted.

# Modulo-2 Arithmetic

However, the addition and subtraction operations that we will use are not regular addition and subtraction.

They are **modulo 2** addition and subtraction.

a + b is actually (a +2 b), i.e., (a + b) mod 2.

In this way, all coefficients are always 0 or 1.

(a⋅x + b) + (a'⋅x + b') = (a +2 a')⋅x + (b +2 b').

This type of + and – of coefficients preserves the basic properties of polynomial arithmetic (e.g., polynomial ⋅ and / distribute over + and -).

# Modulo-2 +/- = XOR

Modulo 2 addition and subtraction are nothing more than bitwise exclusive or (XOR)

E.g.,

1010 1010

+0110 -0110

----- ----

1100 1100

That is, + and - are *both* the same as XOR (there is no carry!)

# Receiver Revisited

* Let n = 4, thus |G| = n+1 = 5, |R| = n = 4 (leading zeroes of R included)
* Sender computes R = remainder of (M\*xn)/G
* Sender sends M;R,
* Receiver computes remainder of (M\*xn - R)/G
* Note: (M\*xn - R) = M;R, because “–“ equals XOR, and |R| = n, i.e.,

M\*x4 = M;0000, (M\*x4 - R) = (M;0000 - R) = M;R

* Thus, the receiver computes (M\*xn - R)/G = (M;R)/G

This is convenient, because the division steps can be performed immediately as each bit of M;R is received.

# Division

Since + and - is not as before, then neither is division. Also, remember that we deal with **polynomials.** The only way to teach it is by example.

x + 1

----------------------

x3 + x2 + x + 1 | x4 +0x3 + x2 + x + 0

-(x4 + x3 + x2 + x)

------------------

x3 +0x2 +0x + 0

-(x3 + x2 + x + 1)

----------------

x2 + x + 1

The above division is written as a string of bits as follows.

**11**

**-----**

**1111 |10110**

**1111**

**----**

**1000**

**1111**

**----**

**111**

Let M = 111001 and G = 11001 (i.e., n = 4). Sender computes (M\*xn)/G

**101101**

**--------------**

**11001 | 1110010000**

**11001**

**-----**

**01011**

**00000**

**-----**

**10110 <---- strange!**

**11001**

**-----**

**11110**

**11001**

**-----**

**01110**

**00000**

**-----**

**11100**

**11001**

**-----**

**0101 = R** (4 (i.e. n) bits, must include leading 0’s)

The receiver computes the following division, whose remainder is zero

(you can carry out the division if you like, I will not bother)

--------------

11001 | 1110010101

|----||--|

M R

# Analysis

Let us know analyze the power of this approach

Let E be the *error mask* caused by the channel, i.e., a bit string indicating which bits got corrupted.

If there were no corrupted bits, E = 0. Otherwise, E  0.

# E Example

If M;R = 110011 and E = 101, then

the bit string received by the receiver is (M;R) + E, i.e.,

(M;R) + E = 110**1**1**0**

i.e. the first and third bits (from right to left got corrupted).

I.e., the receiver receives (M;R) + E, which equals (M;R) XOR E (recall modulo 2).

Thus, M’;R’ = (M;R) + E, which is what the receiver receives.

Of course, the receiver does not know E, it just knows it receives M’;R’ for some M’ and R’.

# Receiver operations

The receiver performs the division with the bits it received

The receiver performs M’;R’/G = (M;R + E)/G and checks the remainder.

If the remainder is zero, it assumes E = 0 and no corruption occurred.

If the remainder is nonzero, you know E  0 and corruption occurred.

# Focus on E

Notice that (M;R + E)/G = (M;R)/G + E/G.

Recall that (M;R)/G has a zero remainder.

Thus, (M;R + E)/G has a nonzero remainder iff E/G has a nonzero remainder.

Thus, to show if corruption errors can be detected, we do not concern ourselves with the original message, we focus on the error mask E.

## Next, for different types of error masks E, we see if the protocol can detect the corruption (i.e., if a nonzero remainder is obtained)

# Single Bit Errors

If a single bit got corrupted, then E is of the form 1000 . . . 00.

I.e., E = xj, for some j.

**To detect corruption, E/G must have non-zero remainder**.

I.e., there should be no polynomial P(x) such that G\*P=E.

Claim: If G has two or more terms, then it cannot divide E

E.g. G could be 10100, 110001, 11, etc.

Thus, G(x) = xb + . . . + xc

Why does it work?

Consider any P(x) = xe + . . . + xf

G(x) \* P(x) =

x^e + . . . + x^f

\* x^b + . . . + x^c

======================

x^(e+c) + . . . + x^(f+c)

. . .

+

. . .

x^(e+b) + . . . + x^(b+f)

===================================

x^(e+b) + . . . . . . . + x^(f+c)

Note: in the sum, x(e+b) appears only once, and x(f+c) also appears only once. Hence, these two terms MUST exist in G(x)\*P(x)

Thus, G(x)\*P(x) cannot be of the form xj for any j.

# Single Bit Errors Example

Consider the following example, with G = 101 and E = 100000

|  |  |
| --- | --- |
| 1010  --------  101 | 100000  101  ---  010  000  ---  100  101  ---  010  000  ---  10 | It does not matter how long E is, E could be 1000000000000000, and still the remainder would be nonzero.  This is because the second 1 bit of G is always "matched" against a zero bit of G, causing a nonzero remainder at each step |

# Double Bit Error

Two bits get corrupted, so E is of the form 10…010…0,

i.e. E(x) = xi + xj for some i > j, i.e., E(x) = (x(i - j) + 1)\*xj

We claim E/G has a nonzero remainder provided *both* of the following conditions hold:

1. G does not divide the string xr + 1 for any r up to the message size
2. G contains the term x0 (i.e. + 1, i.e., the right most bit of G is a 1)

Why?

* From 1 above, G does not divide x(i - j) + 1, so we have a nonzero remainder up to this point in the division.
* Since G has x0, it will continue to generate a nonzero remainder. For all steps in the division.

## Primitive Polynomials

The smallest value of r such that a polynomial divides xr + 1 is known as the *order* or *exponent* of the polynomial.

The polynomials with the highest order are known as *primitive polynomials.*

For polynomials with 0 and 1 coefficients, primitive polynomials of degree n have an order of 2n – 1 (does not divide xr + 1 where 1 < r < 2n).

Thus, if G has a primitive polynomial of degree n as a factor, it can catch all double bit errors in messages up to size 2n bits!

How to find primitive polynomials?

# Odd number of bits

Assume E contains an odd number of corruptions (an odd number of 1's in E).

I.e., if E = 1011, then

E = x3 + x + 1.

Claim: if E has an odd number of terms, then (x+1) is not a factor of E.

**Assume** x+1 is a factor of E. Then, E(x) = (x+1)\*P(x) for some polynomial P.

If we evaluate E at 1, i.e., if we assign 1 to x, then

E(1) = (1+1)\*P(1) = 0\*P(1) = 0 (recall modulo 2 arithmetic)

However, since E has an odd number of terms, then E(1) = 1, due modulo 2 addition. Hence, x+1 ***cannot be a factor of*** E.

Therefore, if *we choose G such that (x+1) is a factor of G*, then E/G has a nonzero remainder because E does not have x+1 as a factor.

# Error burst of k bits, 2 ≤ k ≤ n

In this case, E = 1bbbb…b100…0, where each of the b bits can be either 0 or 1.

I.e., E = xi + … + xj for some i > j and (i - j) ≤ n - 1

E.g., E = 10110010000

i = 10, j = 4

Claim: if G contains the term x0 (i.e. “ + 1” or the rightmost bit of G is 1), then it does not divide E.

Since G has n + 1 bits, it is longer than 1bi1, and (1bi1/G) has nonzero remainder.

Since the remainder after the division steps up to 1bi1 is nonzero, then the remainder at the end of the division is nonzero (next bit dropped is always a 0 which is matched against the 1 in G).

# Error burst of n+1 bits

E and G have the same number of bits.

There is no guarantee that the remainder is nonzero, since if by chance E = G the remainder is obviously zero.

The probability of the remainder being zero ***given*** that the burst size is n+1 (and thus the message is accepted incorrectly) is 1/2n-1

Why 1/2n-1 and not 1/2n+1? The first and last bits don't count, since they are always fixed to be 1.

# Error burst greater than n+1 bits

In this case, the remainder is zero with probability 1/2n.

# (2n possible remainders, you get one randomly)16 bit Standard CRC

The 16 bit standard CRC (n = 16, x16 + x15 + x2 +1), catches:

all single-bit errors,

all double-bit errors,

all odd-number of bit errors,

all bursts of length at most 16,

99.9969% of length 17,

and 99.9985% of length greater than 17

# .IP Checksums (you are not responsible for this)

CRC's are done in hardware (for example, in an Ethernet card), but IP runs in software. Therefore, CRC would be too time consuming. Instead, IP uses a "checksum".

The checksum is based on 1's complement arithmetic (similar, but not identical, to 2's complement).

E.g., let A = 0001 0100 be an 8 bit positive number. -A would be stored as 11101011 (i.e., all bits are flipped)

In 1's complement, we have two zeroes, the regular 0, expressed as 0000 0000, and -0 = 1111 1111.

The rules of adding two numbers in 1's complement are similar (but not identical) to the rules of adding two numbers in 2's complement.

The IP checksum is computed over the IP header (IP does not care if the data is corrupted, that is TCP's problem)

The checksum is a 16 bit field in the header.

To compute it, divide the header into 16 bit words.

Add all the words together using 1's complement arithmetic (similar to 2's complement, but not identical).

The checksum field is the 1's complement of the result.

Notice that taking the 1's complement effectively changes the sign of the result.

The receiver computes the 1's complement addition of all the words in the header (including the checksum), and the result should be 0 (or -0).

For example, if the header words are A,B,C,D, and using 1's complement we get, A + B + C + D = E, then the checksum is -E, and -E is included in the header. Then, the receiver computes A+B+C+D+(-E) = 0.

If the result is not zero the message is corrupted.

There are several reasons why 1's complement was chosen instead of 2's complement, but we will not discuss them here. One of the nice things about this checksum is that, as an IP message moves from router to router, the "time to live" field of the IP message is decreased by 1, and instead of recalculating and updating the checksum at each hop (because the header changed since the TTL field changed), we simply subtract one from the old checksum and we obtain a new and valid checksum for the updated header.