Cyclic Redundancy Checking

Norman Matloff
Dept. of Computer Science
University of California at Davis

February 5, 2001

1 The Importance of Error Detection

A transmitted bit can be received in error, due to "noise" on the transmission channel. If we are dealing with voice or video data, the occurrence of errors in a small percentage of bits is quite tolerable, but in many other cases it is crucial that all bits be received intact. If for example we are downloading a binary program file, the program may be unexecutable if even one bit is incorrect. If the destination or source address of a packet has one bit wrong, communication is impossible.

There are many methods which have been developed to detect errors, applied at different levels of the seven-layer model. Of course, no method can detect all errors, but a number of methods in use today are amazingly effective. The one we will discuss here is the famous CRC coding form.

2 The CRC Error-Detection Method

One powerful error-detection method is Cyclic Redundancy Checking (CRC), a generalization of parity checking. One appends a few (typically 16 or 32) bits to the end of the bit string for a message and sends out the extended string. The receiver then performs a computation which would yield 0 if no bits of the message had been in error; if the result is not 0, then the receiver knows that there has been an error in one or more bits.

CRC cannot detect all possible errors, but the range of errors which it **can** detect is impressively broad. As networks textbook authors Larry Peterson and Bruce Davie have pointed out, it is quite remarkable, for instance, that a mere 32 bits of CRC are sufficient to provide adequate protection against errors in the 12,000-bit messages which Ethernet can send.

3 How CRC Works

Let M be the message we wish to send, m bits long. Let C be a divisor string, c bits long. C will be fixed (say hardwired into a serial I/O chip), while M (and m) will vary. We must have m $>$ c-1. It is always assumed that c $>$ 1 and that the first and last bits of C are 1s.

Our CRC field will consist of a string R, c-1 bits long. Here is how to generate R, and send the message:

- 1. Append c-1 0s to the right end of M. These are placeholders for where the CRC will go. Call this extended string M'.
- 2. Divide M' by C, using mod-2 arithmetic. *Note carefully: This is mod-2, not base-2. These quantities are NOT numbers in the usual sense.*

  Note that each time we divide C, which is a c-bit quantity, into another c-bit quantity Z, we say that C "goes into" Z 1 time if the leading bit of Z is 1, or 0 times otherwise.

  Call the remainder R. Since we are dividing by a c-bit quantity, R will be c-1 bits long.
- 3. Replace the c-1 appended 0s in M' by R. Call this new string W.
- 4. Send W to the receiver.

For example, say C = 1011 and M = 1001101. Then we divide as follows:

```
          1010011
       ----------
1011 ) 1001101000
       1011
       ----
         1010
         1011
         ----
           1100
           1011
           ----
             1110
             1011
             ----
              101
```

We first divide 1011 into 1001 (see above), with a quotient of 1. Yes, that's right! As long as the dividend has at least as many digits as the divisor–both have 4 digits here–then the divisor does indeed "go into" the dividend with a quotient of at least 1. If they have the same number of digits,[1] the quotient is 1, which is placed as the first digit in the overall quotient above the division sign. Our remainder is 0010 (remember, our arithmetic here is mod 2, bitwise), which we write as 10.

Next we extend that 10 by bringing down the next digit in the dividend, a 1, forming 101, that is 0101. 1011 goes into 0101 0 times, so we place this 0 as the next digit in the overall quotient above, and bring down the next digit of the dividend, a 0, making 1010. Now 1011 goes into 1010 1 time, and so on.

[1]The number of digits is counted beginning with the first 1 digit, so for example 0011 is considered only a 2-digit quantity, not a 4-digit one.

The final remainder is 101, so the transmitter now sends the string 1001101101.

(You should check that one can "multiply" back to get the original result, i.e. 1011*1010011 + 101 = 1001101000. Remember, though, that the multiplication and addition are done on a mod-2 basis, i.e. XOR with no carries.)

The receiver then receives a string Y, which is hopefully the same as W but might not be, due to line noise. The receiver divides Y by C; if the remainder is nonzero, the receiver decides Y had one or more bits in error, while if the remainder is zero, the receiver accepts Y as correct (though it still might not be).

4 Why It Works

Note that the replace operation in step 3 above is equivalent to performing

$$W \leftarrow M' + R \tag{1}$$

Moreover, because we are using mod 2 arithmetic, that operation is also equivalent to

$$W \leftarrow M' - R \tag{2}$$

If one divides, say, 63 by 5, then one gets a remainder of 3. That means that if we subtract that remainder of 3 from the dividend 63, the result 60 will be evenly divisible by 5.

Similarly, since R was our remainder from M'/C, we see from the last equation that M'-R is exactly divisible by C. So, W is divisible by C.

Let E denote the **error vector**, a symbolic way to describe which bits in W get corrupted on their way to the receiver. If for instance the i-th bit of E is a 1, this means that there was an error in the i-th bit of W. Then the receiver will receive Y = W + E, again keeping in mind that this is mod-2, bitwise addition.

We hope E consists of all 0s, of course, but it might not. The receiver will divide Y by C; since W is evenly divisible by C, then Y will be too if E = 0. So, if the receiver finds that Y/C has a nonzero remainder, the receiver can be sure that there was an error.

In other words, the CRC will detect all error patterns except those for which E is evenly divisible by C. Clever people have found good strings C for which a remarkably small proportion of Es are evenly divisible by C. These Cs have been publicized and are in common use.

Note that E is an m+(c-1) bit quantity, so it includes the bits corresponding to the R portion of W. So, the system here is even capable of detecting some errors in the CRC field itself!

By the way, when we divide M' by C, getting a quotient Q and a remainder R, Q will be m bits long. To see this, draw a "long division" diagram as above, and note that the first and last bits of Q will be above the c-th and (m+c-1)-st bits of M', so that Q consists of (m+c-1) - c + 1 = m bits.

5 What Kinds of Errors Will CRC Detect?

In CRC analysis, it helps to represent bit strings as polynomials. For instance, our C above, 1011, would be denoted by

$$x^3 + x + 1 \tag{3}$$

The word “denoted” here is key; the polynomial is just notation, and there is no physical variable x. But this notation turns out to be an extremely powerful tool for analyzing CRC. We will use C(x), E(x) and so on to refer to the polynomial versions of the bit strings C, E etc.

The question at hand is which types of E strings are detectable by which C strings. Remember, we decide what string to use for C, say when we design a serial I/O chip or Ethernet interface card. So, we want to choose C in such a way that many classes of E strings are detectable. Recall that an E string will be detected as long as C does not evenly divide it, and our analysis will stem from this consideration.

Following are two examples of the many properties CRC theorists have proven.

- **Any single-bit error will be detected.**

  To see this, note that for such an error pattern, E will consist of all 0s except for a single 1 bit in some position within the string. So, E(x) will be x^k for some k.

  Recall that it is always assumed that c $>$ 1 and that the first and last bits of C are 1s. In other words, C(x) always has more than one term, and always has 1 as its constant term. But it would be impossible for such a C(x) to divide x^k, for the following reasons.

  If k is greater than 0, then if C(x) were to evenly divide x^k, that would imply that x is a factor of C(x), which it is not, due to the presence of the nonzero constant term in C(x). If k is 0, so that E(x) = 1, then the fact that C(x) has at least one other term than 1 means C(x) could not divide E(x).

  So, this E(x) is guaranteed not to be evenly divisible by C(x), and thus a single-bit error will be caught by the CRC.

- **Any burst error, i.e. a set of consecutive bit errors, of length at most c-1 will be detected.**

  In our worked-out example above in which W = 1001101101, if E is, say 0000111000, with consecutive errors in the fifth, sixth and seventh bits of W, then such a pattern of errors will be detected.

  To see this, let w denote the length of W, which is also the length of E and Q. Say the most significant 1 in E is in bit position j, where the least significant position is called position 0 and the most significant position is numbered w-1. Denote the length of the burst by b. Then E(x) will have the form

$$x^j + x^{j-1} + x^{j-2} + ... + x^{j-b+1} \tag{4}$$

  which can be factored as

$$x^{j-b+1}(x^{b-1} + ... + 1) \tag{5}$$

Suppose C(x) were to evenly divide this E(x). Then C(x) would have to evenly divide either x^{j-b+1} (which is already in "prime factorization" form) or $x^{b-1} + ... + 1$. The former case is impossible, as we saw above. The latter case is also impossible, since b was given to be at most c-1 and thus $x^{b-1} + ... + 1$ has degree less than that of C(x).

So we see C(x) cannot evenly divide E(x), and our claim is proved.

This property of catching burst errors is quite important, as it is often the case that line noise will indeed occur in bursts.

6 Digital Implementation

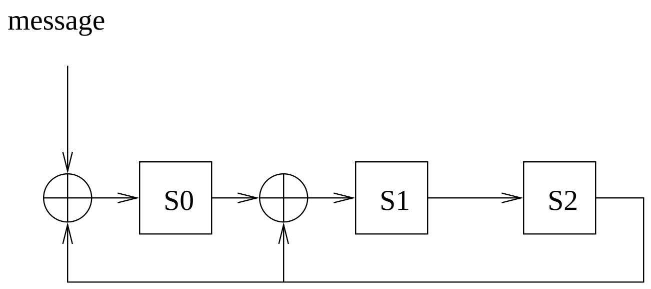
As you can see, the CRC algorithms could easily be encoded in software, but that may be too slow. It would be much faster if implemented in digital logic hardware, and it turns out that this can be done quite simply.

As before, let c denote the number of bits in the divisor polynomial C(x), so that the degree of the polynomial is c-1. Then we set up a left-to-right shift register with c-1 bits, which we will denote, from left to right, $S_0, S_1, ..., S_{c-2}$.. As with any shift register, at each clock impulse S_i is replaced by the value which had been in S_{i-1}, where we are calling the left input "S_{-1}."

However, our shift register will differ from an ordinary one in that at each clock pulse the value which had been in S_{c-1} will be recycled and XOR-ed with some of the other S_i. Specifically, if there is a term x^j in the divisor polynomia C(x), then there is an XOR just to the left of S_i (except for i = c-1).

The message M, with c-1 0s appended as before, is fed into the input to the shift register, one bit per clock cycle, starting with M's most significant (i.e. leftmost) bit. After m+c clock cycles, the shift register contains the remainder (i.e. the CRC field), in reverse order.

Our figure here shows the setup for the case $C(x) = x^3 + x + 1$, i.e. C = 1011, from our example above.



The table here shows a pulse-by-pulse account of what will occur with the input M = 1001101 (with three 0s appended):

clk	input	S0	S1	S2
0	1	0	0	0
1	0	1	0	0
2	0	0	1	0
3	1	0	0	1
4	1	0	1	0
5	0	1	0	1
6	1	1	0	0
7	0	1	1	0
8	0	0	1	1
9	0	1	1	0
end		1	0	1

For example, at the end of clock period 3, the register contains (0,0,1), with an input of 1 coming in, say, near the end of the clock period, taking effect in the next period. Then the clock pulse starting period 4 comes. What happens?

- The new S_0 will be the XOR of 1 (from the old S_2) and 1 (from the input shown in row 3), which is 0. Thus the new S_0 will be 0.
- The new S_1 will be the XOR of 1 (from the old S_2) and 0 (from the old S_1), i.e. 1.
- The new S_2 will be equal to the old S_1, i.e. 0

So, the new register state will be (0,1,0), seen in row 4.

This would occur at the transmitter end, and a similar setup for the CRC check will be at the receiver end.