Week 3: Introduction - Error detection

Error detection and correction

When transferring data within a computer, for example when writing data to disk, or from one computer to another using a communication line, errors can occur due to noisy lines or simply malfunction. These errors may cause some 1s to become 0s and some 0s to become 1s, as in:

"**ABC**" = 1000001 1000010 1000011

when transmitted to another computer or stored may become:

1000001 1000010 100**1**011 = "**ABK**"

In data communications if an error is detected it may be fixed by requesting retransmission, but when writing data to disk that this may be more difficult.

To solve that problem, extra *redundant bits* (also called *check bits*) are added to the original information before transmission. As a first example, let's add an extra *parity* bit to a 7-bit ASCII character to detect an error. Depending on the number of 1s that we choose, even or odd parity may be used; there are no advantages of one over the other, so they are used indistinctly.

Even parity

For ASCII characters, bit 7 (that is, the eight-bit, the most significant bit) is set so that the total number of 1s in a character is even, as in:

'A' = 100 0001 Parity = 0 Code = **0**100 0001
Parity Bit = 0

'C' = 100 0011 Parity = 1 Code = **1**100 0011
Parity Bit = 1

Odd parity

The parity bit is set so that the total number of 1s is odd, as in:

'A' = 100 0001 Parity = 1 Code = **1**100 0001
Parity bit = 1

If we consider for example 'A' with even parity 0100 0001, any 1-bit error will change the number of 1s and 0s in the symbol, and thus, the parity of the symbol. Say the error is on bit 3, so the symbol received is 0100 1001; this error is easily detected because the number of 1s is 3 (odd) instead of an even number.

Caution

A single parity bit can only detect an error, it cannot be used to correct errors; we need more redundant bits if we want to also correct errors.

Error correcting codes

Simple parity is a particular example of an error detecting code. We have just mentioned that although using simple parity it is possible to detect errors, it is not possible to correct them.

Say we want to send a message: Yes = 1, No = 0. If there is an error in transmission, the receiver wouldn't know because they would not be able to detect a 0 changed into a 1, or a 1 into a 0. If instead we use an extra parity bit, and we agree that the number of 0s must be even, for example, to send a 0 we send 00 and to send a 1 we send 11. If there is a one-bit error, the receiver will get 01 or 10, and they will know that there has been an error. However, they will not be able to find out what was the original message.

If instead we use 3 bits to send our message, our messages are always either No = 000 and Yes = 111. If we assume that there is only a one-bit error and we receive 001, what symbol was originally sent? Obviously 000, because 111 has more than one bit difference with 001, so that cannot be the source of a one-bit error. Thus, since there are 3 bits difference between 000 and 111, we find the mistake and correct it.

An *error correcting code* is able to detect and correct bit errors in a (binary) code. In 7-bit ASCII code, a 1-bit change to a valid symbol gives another valid symbol. It is then not possible to detect or correct a one-bit error, because a 1-bit error is still a legal symbol. Adding a parity bit made it possible to detect, but not correct, an error. To detect or correct errors, codes use extra bits to increase the distance between valid symbols, as in:

1111 1000 -> 111 1001 -> 1111 1011 -> 1111 1111
valid..... invalid.... invalid..... valid

Hamming distance

In a code, the number of bit changes between two valid symbols of the code is called *Hamming Distance*. In the example, the Hamming Distance has been made constantly 3, and therefore 3 bits must change to arrive at the next valid symbol. Thus, if one bit error is detected, it may be corrected by selecting the closest valid symbol.

To detect up to K bit errors, the (minimum) Hamming distance between two valid code words must be K+1. For example, simple parity makes 2 the distance between symbols and we have seen that it detects a one-bit error in ASCII. To correct up to K bit errors, the Hamming distance between two valid code words must be greater than or equal to 2K+1.

Hence, to detect and correct 1 error requires a Hamming distance of 3.

Hamming codes

In the general case, the way we add those bits is up the scheme used. The most popular one is the *Hamming code*.

In the Hamming code the number of Parity/Check bits p must be:

$2^p \geq m + p + 1$

where m = number of data bits and p = number of check bits.

An example:

Find the number of check bits required to detect and correct a single bit error in the BCD code for $9_{10} = 1001_2$

As a guess we can try p = 2 then $2^p = 2^2 = 4$, but since 4 + 2 + 1 = 7 and 4 is not ≥ 7, p = 2 is not enough.

However, if we now try p = 3 then $2^p = 2^3 = 8$ and m + p + 1 = 4 + 3 + 1 = 8. Therefore p = 3 is sufficient.

A Hamming code includes several parity bits, to be able to detect and correct 1-bit errors. In principle the parity bits may go in many different places, but the most common use is interspersed with the data bits, because:

- to be able to put them at the beginning or the end we have to know in advance the number of data bits, and that it not always possible;
- giving them fixed positions means that we know exactly where the extra bits are.

To get the position of the check bits we count the bits from right to left (i.e. LSB to MSB) starting from 1, and the check bits are at positions 1, 2, 4, 8, 16, 32, etc, all powers of 2. All the other bits are data bits, part of the message.

Bit Position	10	9	8	7	6	5	4	3	2	1
Data / Parity	D6	D5	P4	D4	D3	D2	P3	D1	P2	P1

The table above is an example of Hamming code for 6 data bits (P = parity bit, D = data bit).

The scheme works by using each check bit as a parity bit for a specific group of bits, as follows:

- *P1 covers bits 1, 3, 5, 7, 9, 11, 13, 15, etc.*
  P1 covers positions where there is a 1 in the first binary bit of the position number binary expression (highlighted in boldface): 000**1** , 001**1** , 010**1** , ...
- *P2 covers bits 2, 3, 6, 7, 10, 11, 14, 15, 18, 19, etc.*
  P2 covers positions where there is 1 in the second binary bit of the position number binary expression:
  00**1** 0, 00**1** 1, 01**1** 0, 00**1** 1, ...
- *P3 covers bits 4, 5, 6, 7, 12, 13, 14, 15, etc.*
  P3 covers positions where there is a 1 in the third binary bit of the position number binary expression:
  0**1** 00, 0**1** 01, 0**1** 10, 0**1** 11, ...
- *P4 covers bits 8, 9, 10, 11, 12, 13, 14, 15, etc.*
  P4 covers positions where there is a 1 in the fourth binary bit of the position number expression:

1 000, **1** 001, **1** 010, **1** 011, **1** 100, **1** 101, **1** 110, **1** 111

We can think about it in a different way:

- **P1** covers bits 1, 3, 5, 7, 9, 11, 13, 15, etc.
  P1 covers one bit, skips one bit
- **P2** covers bits 2, 3, 6, 7, 10, 11, 14, 15, 18, 19, etc.
  P2 covers two bits, skips two bits
- **P3** covers bits 4, 5, 6, 7, 12, 13, 14, 15, etc.
  P3 covers four bits, skips four bits
- **P4** covers bits 8, 9, 10, 11, 12, 13, 14, 15, etc.
  P4 covers eight bits, skips eight bits

An example: determine the single bit error-correcting Hamming code for the data 1011_2.
Use EVEN parity. From the formula we can see that p = 3 is sufficient, so we can write:

7	6	5	4	3	2	1
1	0	1		1		

Therefore the final code is 1 0 1 **0** 1 **0 1**.

- P1 checks bits 1, 3, 5, 7 = ? 1 1 1, P1=1
- P2 checks bits 2, 3, 6, 7 = ? 1 0 1, P2=0
- P3 checks bits 4, 5, 6, 7 = ? 1 0 1, P3=0

Therefore the final code is 1 0 1 **0** 1 **0 1**.

The idea is that the parity bits cover each bit (data and parity) with a unique pattern, that is each bit (including the parity bits) is covered by a unique combination of parity bits. For example, bit 5 is covered by bits P1 and P3 only and exclusively, and bit 4 is covered by P3 exclusively.

Then, if we check and find a parity error on bits P1 and P3, but no other parity errors, we know that the problem is with bit 5. So, if bit 5 is a 1, we change it to 0, and if it is a 0 we change to 1 to correct the symbol.

Why only one-bit errors?

It may be surprising that we are only interested in 1-bit errors, but in computer operation 2-bit errors are very, very unlikely. If the probability of a 1-bit error is of the order of 10-9, that is 1/1,000,000,000, this is really small, but if a computer makes 10,000,000 moves a second, on average you get an error every 100 seconds = less than 2 minutes.

However, since these communications are most likely parallel, 2-bit errors occur when there is one error on two lines at the same time. That is, the probability is the product of the two probabilities 10-18. With the same computer you get a 2-bit error once every 1011 seconds = once every 3,171 years.

There may be other considerations, specifically to do with data communications. It is quite common to establish communications over noisy lines, for example, and then the probability of errors increase dramatically. It often happens that there is a short period when may be multiple-bit errors, and it would be impracticable to use a Hamming code in this situation. Other schemes more appropriate to this problem are used instead.

SECDED coding

The ability of a Hamming code to correct an error is based on the assumption that only one bit has changed. Changes of two or more bits either are not going to be detected, or will give a false indication of which bits are in error. A common scheme to improve on the situation is called *SECDED Coding* (single-error correct, double-error detect), which makes possible the correction of single-bit errors and the detection, but not correction, of double-bit errors.

The original Hamming code is extended by using the unused P0 bit as a parity (odd or even) bit for all the bits in the transmitted symbol. In this way, if one bit is in error, the Hamming checks will fail for a particular set of bits, and also the overall parity will be wrong.

We may then assume that there has been a 1-bit error and the Hamming code can correct it. If 2 bits are in error, some Hamming checks will fail but the overall parity will be right, indicating that more that 1 bit was in error but this time we are not able to correct it.

Summarising, using SECDED coding:

- If some Hamming checks fail, and there is also an overall parity error, we assume that there was a 1-bit error and we correct it as before.
- If some Hamming checks fail but there is no overall parity error, we assume that there have been 2 or more errors, and we don't correct them.

Of course, is there have actually been more than 2 errors, the SECDED scheme will give a wrong result. As we have discussed, this is very unlikely.