Error detection and recovery

Errors occur because the signal is corrupted during its trip from the transmitter to the receiver. While some errors can be detected immediately by the receiver (PL), most result in the signal being incorrectly decoded.

Besides jamming the transmitter, the PL has no means of telling the transmitter that a frame was not received correctly (and even jamming does not guarantee that the transmitter will take notice).

The typical arrangement is that the PL passes a complete frame to the DLL (even if it clearly is incorrect) and it is the DLL's task to validate it.

Validation is based on two tests:

Header: the body of the frame must correspond to the header of the frame (especially the length).

Checksum: the checksums calculated for the received frame must match the checksums sent together with the frame.

The term checksum is acquiring a narrower meaning: to denote a non–cyclic checksum; cyclic checksums are increasingly called **CRC** and not considered “checksums” anymore.

Detection and correction

A first step is to determine whether the transmission was error–free. If it was not, three possibilities exist:

- Discard the frame. This works if the communication protocol requires positive acknowledgments (the transmitter will retransmit the frame) or in real–time communication (there is no use for another copy of the frame).
- Ask for retransmission (negative acknowledgment). This is called **backward error correction**.
- Attempt to correct the frame (and, if not successful, resort to one of the other two options). This is **forward error correction (FEC**).

Note that a frame is deemed error–free is it passes a statistical test which is not the same as containing no errors.

Redundant bits

The simplest idea is to treat the whole frame as one unit.

The sender adds to the transmitted frame a sequence of bits that are a coding function of the contents of the frame. That is, a frame of *n* bits is interpreted as one very long *n*–bit integer in binary notation $\mathcal{F}$ and a well–chosen function C is applied to it. The result is $C(\mathcal{F})$, an integer which can be viewed as a sequence of *r* bits.

Thus, instead of sending the frame $\mathcal{F}$ of *n* bits the transmitter sends the *n*+*r* bit sequence $\mathcal{F}c$ where $c = C(\mathcal{F})$ is the redundant part of the frame.

The receiver sees this sequence as $\mathcal{F}_r c_r$.

The receiver performs the computation of $C(\mathcal{F}_r)$ and compares the result with c_r. If the two match, the frame is assumed intact. Otherwise, the frame contains bit errors.

If C was chosen well, it can give us hints about the location of the error(s) which leads to a potential FEC.

Block coding

The method used in practice applies the above idea not to whole frames (which are of variable lengths) but to fixed–size chunks of frames.

The frame is divided into blocks of fixed size (*n* bits) (the last block is padded if too short). A suitable function C is used; this function takes an *n* bit integer as argument and produces an *r* bit result.

The frame block of length *n* is called a **dataword** while the corresponding sequence of *n*+*r* bits is called a **codeword**.

One immediate observation: since we construct codewords out of legal datawords only, there can be only 2^n legal codewords. There are 2^{n+r} codewords in all, hence a randomly chosen codeword is legal with probability $\frac{1}{2^r}$.

Some math

Consider arithmetic with 0 and 1 being the only numbers allowed (no multi–digit numbers). It is called arithmetic *modulo 2* (because it gives the same result as "normal" arithmetic with only the remainder from division by 2 kept).

$\oplus$ is the addition and $\ominus$ the subtraction symbol for this arithmetic:

Operation	Result
$0 \oplus 0$	0
$0 \oplus 1$	1
$1 \oplus 0$	1
$1 \oplus 1$	0

Note that $\ominus$ is the same as $\oplus$:

Operation	Result
$0 \ominus 0$	0
$0 \ominus 1$	1
$1 \ominus 0$	1
$1 \ominus 1$	0

2–bit parity block code

A simplistic 2–bit block code: $n = 2$ and C is the sum of bits using $\oplus$ (hence $r = 1$).

Dataword	C	Codeword
00	0	000
01	1	011
10	1	101
11	0	110

All the legal codewords in this simple code have an even number of 1s. This is sufficient to detect all single–bit errors within one block (parity check).

Hamming distance

Consider two binary strings $\alpha = \{\alpha_i\}$ and $\beta = \{\beta_i\}$ (both strings must have the same length).

The **Hamming distance** *d* between α and β is defined as the number of bits where $\alpha_i \neq \beta_i$.

More formally:

$$d(\alpha, \beta) = \sum_{i=1}^{n} \alpha_i \oplus \beta_i$$

where the summation is a simple addition and *n* is the length of the strings.

Minimum Hamming distance

Consider a set of codewords $W = \{W_i\}_{i=1..n}$. The minimum Hamming distance d_{min} for this set is the smallest distance between all pairs of codewords in the set:

$$d_{min}(W) = \min_{i=1..n, j=1..n, i \neq j} d(W_i, W_j)$$

For the 2–bit parity block code, $d_{min} = 2$ because $d(1,2) = 2$, $d(1,3) = 2$, $d(2,3) = 2$ and $d(3,4) = 2$. This is true for any *n*–bit parity block code.

Error detection and Hamming distance

A block code with $d_{min} = s + 1$ guarantees the detection of all errors within the block involving no more than s bits.

Examples:

- All errors of 1 bit or 2 bits can be detected in a block code with $d_{min} = 3$. Some errors involving more bits might also be detectable, but not all.
- Any *n*–bit parity code has $d_{min} = 2$. Hence single–bit errors can be detected. In fact all errors involving an odd number of bits (within a single block) can be detected.

Correction of errors

A block code with $d_{min} = 2s + 1$ guarantees the correction of all errors within the block involving no more than s bits.

This implies that block codes with an odd minimum Hamming distance should be considered more efficient that those with an even d_{min}.

Polynomial arithmetic

Again, the only allowable values are 0 and 1.

A polynomial $P(x)$ of degree *n* is defined as:

$$P(x) = \sum_{i=0}^{n} a_i x^i$$

where $a_i \in \{0, 1\}$.

Note that a polynomial of degree *n* can be viewed as a pattern of *n+1* bits that looks identical but should not be confused with a binary–encoded integer which also is, in fact, a polynomial ($x = 2$) similarly represented.

The difference is in the way operations on them are defined.

Polynomial addition

The sum of two polynomials $P_1(x) = \sum_{i=0}^{n} a_i x^i$ and $P_2(x) = \sum_{i=0}^{m} b_i x^i$ is (assuming that $n \geq m$):

$$P_1(x) \oplus P_2(x) = \sum_{i=0}^{n} (a_i \oplus b_i) x^i$$

$$(x^6 + x^2 + 1) \oplus (x^5 + x^2 + 1) = x^6 + x^5$$

In bit representation:

$$1000101 \oplus 0100101 = 1100000$$

while

$$1000101 + 0100101 = 1101010$$

Polynomial subtraction is the same as addition.

Polynomial multiplication

Polynomials are multiplied term by term. Then the coefficients are summed using $\oplus$.

$$
\begin{aligned}
(x^6 + x^3 + x) \otimes (x^2 + x + 1) = \quad & x^8 + x^5 + x^3 \\
& \oplus \\
& x^7 + x^4 + x^2 \\
& \oplus \\
& x^6 + x^3 + x
\end{aligned}
$$

$$= x^8 + x^7 + x^6 + x^5 + x^4 + x^2 + x$$

Polynomial division

Division is done in a manner similar to “normal” division but is a bit easier.

When $P(x) = \sum_{i=0}^{n} a_i x^i$ of degree n is divided by $Q(x)$ of degree r, a sequence of $n - r + 1$ operations is performed.

- Each operation subtracts $Q(x) \otimes x^i$ from $P(x)$ if the leading bit of $P(x)$ is a 1
  (for $i = n - r, n - r - 1, ..., 0$).
- If the subtraction was performed, a x^i is appended to the quotient.

At the end we have an $n - r + 1$ bit quotient and an r bit remainder.

Division

$P(x) = x^6 + x^5 + x^2 + x$ ($n = 6$) is divided by
$Q(x) = x^3 + 1$ ($r = 3$).

$Q(x) \otimes x^3$

$$x^6 + x^5 + x^2 + x \ominus x^6 + x^3 = x^5 + x^3 + x^2 + x$$

and the quotient is x^3 (and counting).

$Q(x) \otimes x^2$

$$x^5 + x^3 + x^2 + x \ominus x^5 + x^2 = x^3 + x$$

and the quotient is $x^3 + x^2$ (and counting).

$Q(x) \otimes x^1$

$$x^3 + x \ \ vs. \ \ x^4 + x$$

no subtraction is performed in this step.

$Q(x) \otimes x^0$

$$x^3 + x \ominus x^3 + 1 = x + 1$$

and the quotient is $x^3 + x^2 + 1$.

The result is $x^3 + x^2 + 1$ with a remainder of $x + 1$.

Cyclic Redundancy Codes

A **CRC** code is one computed using polynomial calculations.

Given is a set of n bit datawords $W = \{W_i\}$. We want to create a corresponding set of $n + r$ bit codewords that form a **CRC** code.

We need a polynomial of degree r. We will use this polynomial to divide datawords and append the division remainders to the datawords to form codewords.

CRC

Given is a block code of size n (i.e. each dataword is n bits long). We want to have **CRC** codewords of length $n + r$ so we pick a suitable polynomial of degree r called generator $G(x)$.

To compute the codeword for dataword W the algorithm does the following:

- W is interpreted as a polynomial $W(x)$ and is multiplied by x^r. Let $D(x) = W(x) \otimes x^r$.
- $D(x) \oslash G(x)$ is performed and the remainder is stored in $R(x)$. $R(x)$ (which must have r bits) is added to $D(x)$ to form a codeword.

Example: $n = 4$, the dataword is 1001 ($W(x) = x^3 + 1$) and $G(x) = x^3 + x + 1$ (implies $r = 3$).

$D(x) = W(x) \otimes x^3 = x^6 + x^3$ and

$$x^6 + x^3 = (x^3 + x + 1) \otimes G(x) \oplus (x^2 + x)$$

so the remainder of $D(x) \oslash G(x)$ is $x^2 + x$ and the codeword is:

$$x^6 + x^3 + x^2 + x$$

or 1001110 as a sequence of 4+3 bits.

Beauty of $\ominus$

The codeword $D(x) \oplus R(x)$ was produced as after calculating that: $D(x) = Q(x) \otimes G(x) \oplus R(x)$ (where $Q(x)$ is the quotient).

The codeword equals $D(x) \oplus R(x) = Q(x) \otimes G(x) \oplus R(x) \oplus R(x)$. It so happens that the result of $\oplus$ is the same as the result of $\ominus$ so $R(x) \oplus R(x) = 0$ for any $R(x)$ and

$$D(x) \oplus R(x) = Q(x) \otimes G(x)$$

which means that the codeword is divisible by the generator without a remainder.

CRC and error detection

When a codeword sent as $C(x)$ is received, it has the form $C(x) + E(x)$ where $E(x)$ is the error introduced during transmission.

Detection consists of determining whether $E(x) = 0$.

Dividing $C(x) + E(x)$ by $G(x)$, one gets a remainder (ignore the quotient) which is equal to:

$$remainder(E(x) \oslash G(x))$$

because $C(x)$ is divisible by $G(x)$.

The **CRC** test:

> A received codeword is correct if it is divisible by $G(x)$.

Implication: errors divisible by $G(x)$ are not caught by the CRC test; all other errors are caught.

A good generator

- Any generator with at least 2 terms will catch all one–bit errors.

  Proof: a single–bit error has $E(x) = x^k$ for some k.
  No polynomial of the form $x^i + x^j$ divides x^k as long as $i \neq j$.

- A polynomial with an odd number of terms cannot be divisible by $x + 1$.

  Proof: the subtraction step in $\oslash$ division subtracts $x^{i+1} + x^i$; it reduces the number of terms by either 2 or 0, keeping it odd, hence never equal to 0 (terms).

  Moral: a generator divisible by $x + 1$ will catch all errors involving an odd number of bits.

- Any generator of degree r will detect all burst errors (errors in consecutive bits) involving no more than r bits.

Several other interesting observations can be made.

Some useful polynomials

Name	Polynomial
CRC–8	$x^8 + x^2 + 1$
CRC–16	$x^{16} + x^{12} + x^5 + 1$
CRC–32	$x^{32} + x^{26} + x^{22} + x^{16} + x^{12} + x^{11}$ $+x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$

1's complement arithmetic

In *modulo–n* binary arithmetic the result can be computed in two different ways:

2's complement notation, i.e. the result is the remainder after truncating the carry bits. For example,
$1010 \oplus_{16} 1001 = 1 \mid 0011 =_2 11$
$(10 + 9 \ \mathbf{mod}_{16} = 3)$.

1's complement notation, with the result being equal to the remainder plus the carry. The same example:
$1010 \oplus_{16} 1001 = 1 \mid 0011 =_1 100$
(hmmm, $10 + 9 = 4$? Yes, $10 + 9 \ \mathbf{mod}_{15} = 4$).

1's and 2's complement exist to express negative numbers. Humans (and computers) use 2's complement arithmetic, mainly because it has only one zero (in 1's complement there are two zeros: $+0$ and -0).

Traditional checksum

A checksum was originally defined as the sum of all the datawords.

The transmitter divides the frame into n bit datawords. All the datawords are added using **1's complement** *modulo–n* arithmetic. The result, called **checksum**, is then negated and appended to the frame.

The receiver also computes the checksum over all the bits it received (sender's checksum included). The result must be -0 if there are no transmission errors; if there are errors, it might still be -0 (with a tiny probability).

Checksum and the Internet

Checksums are used by several protocols, most notably by:

IP: uses a 16–bit checksum (i.e. $n = 16$).

UDP: also uses an optional 16–bit checksum. The existence of 2 zeros is cleverly used: if the checksum field is a +0 (all zeros), it means that no checksum was sent (if a checksum of 0 was sent, it must have been encoded as the other zero: -0 which is all ones).

Error correction

The methods used nowadays are fairly complex. They are based on complex block codes or on convolution(al) codes which are an extension of the polynomial approach to block codes.

The most common is the Reed–Solomon block code. In its variant used in CDs, DVDs (and many other applications) uses datawords of size 223 bytes which it converts into 255–byte codewords. The (223,32) variant allows to correct any 16 byte–errors per block.

Error correction algorithms require guesswork, because they attempt to reconstruct the most probable legal codeword from a given illegal codeword. The classic (and standard) algorithm is Viterbi's.