Detecting and Correcting Errors

- Codewords and Hamming Distance
- Error Detection: parity
- Single-bit Error Correction
- Burst Error Correction
- Framing
There’s good news and bad news...

The good news: Our digital modulation scheme usually allows us to recover the original signal despite small amplitude errors introduced by the components and channel. An example of the digital abstraction doing its job!

The bad news: larger amplitude errors (hopefully infrequent) that change the signal irretrievably. These show up as bit errors in our digital data stream.
Channel coding

Our plan to deal with bit errors:

We’ll add redundant information to the transmitted bit stream (a process called *channel coding*) so that we can detect errors at the receiver. Ideally we’d like to correct commonly occurring errors, e.g., error bursts of bounded length. Otherwise, we should detect uncorrectable errors and use, say, retransmission to deal with the problem.
More good news, bad news...

Shannon’s Noisy Channel Coding Theorem:

*Given a noisy channel with channel capacity C and information transmitted at a rate R, then if R < C, there exists a code that allows the probability of error at the receiver to be made arbitrarily small.*

- **Good news:** theoretically it is possible to transmit information without error at any rate below the limiting rate C.

- **Bad news:** the proof doesn’t tell how to construct the appropriate error-correcting code for a given R and C!
Suppose we wanted to reliably transmit the result of a single coin flip:

Heads: “0”  
Tails: “1”

Further suppose that during transmission a single-bit error occurs, i.e., a single “0” is turned into a “1” or a “1” is turned into a “0”.

This is a prototype of the “bit” coin for the new information economy. Value = 12.5¢
Hamming Distance
(Richard Hamming, 1950)

HAMMING DISTANCE:
The number of digit positions in which the corresponding digits of two encodings of the same length are different

The Hamming distance between a valid binary code word and the same code word with single-bit error is 1.

The problem with our simple encoding is that the two valid code words (“0” and “1”) also have a Hamming distance of 1. So a single-bit error changes a valid code word into another valid code word…

“heads” 0 → 1 “tails”

I wish he’d increase his hamming distance
Error Detection

What we need is an encoding where a single-bit error doesn’t produce another valid code word.

We can add single-bit error detection to any length code word by adding a parity bit chosen to guarantee the Hamming distance between any two valid code words is at least 2. In the diagram above, we’re using “even parity” where the added bit is chosen to make the total number of 1’s in the code word even.

Can we correct detected errors? Not yet...
Parity check

• A parity bit can be added to any length message and is chosen to make the total number of “1” bits even (aka “even parity”).

• To check for a single-bit error (actually any odd number of errors), count the number of “1”s in the received word and if it’s odd, there’s been an error.

0 1 1 0 1 0 0 1 0 1 0 0 1 1 → original word with parity
0 1 1 0 0 0 0 1 0 0 1 1 → single-bit error (detected)
0 1 1 0 0 0 1 1 1 0 0 1 1 → 2-bit error (not detected)

• One can “count” by summing the bits in the word modulo 2 (which is equivalent to XOR’ing the bits together).
Other ways to detect errors

- **Checksum:**
  - Add up all the message units and send along sum
  - Adler-32: two 16-bit mod-65521 sums A and B, A is sum of all the bytes in the message, B is the sum of the A values after each addition.

- **Cyclical redundancy check (eg, CRC-16)**
  
  ![Diagram of CRC-16](http://www.erg.abdn.ac.uk/users/gorry/course/dl-pages/crc.html)

  Sending: Initialize all D elements to 0. Set switch to position A, send message bit-by-bit. When complete, set switch to position B and send 16 more bits.

  Receiving: Initialize all D elements to 0. Set switch to position A, receive message and CRC bit-by-bit. If correct, all D elements should be 0 after last bit has been processed.

  CRC-16 detects all single- and double-bit errors, all odd numbers of errors, all errors with burst lengths < 16, and 99.984% of all other errors.
By increasing the Hamming distance between valid code words to 3, we guarantee that the sets of words produced by single-bit errors don’t overlap. So if we detect an error, we can perform error correction since we can tell what the valid code was before the error happened.

- Can we safely detect double-bit errors while correcting 1-bit errors?
- Do we always need to triple the number of bits?
Error Correcting Codes (ECC)

Basic idea:
- Use multiple parity bits, each covering a subset of the data bits.
- No two message bits belong to exactly the same subsets, so a single-bit error will generate a unique set of parity check errors.

\[
P_1 = D_1 \oplus D_2 \oplus D_4 \\
P_2 = D_1 \oplus D_3 \oplus D_4 \\
P_3 = D_2 \oplus D_3 \oplus D_4
\]

Suppose we check the parity and discover that \(P_2\) and \(P_3\) indicate an error? 
bit \(D_3\) must have flipped

What if only \(P_3\) indicates an error? 
\(P_3\) itself had the error!
(n, k) Block Codes

- Split message into \( k \)-bit blocks
- Add \((n-k)\) parity bits to each block, making each block \( n \) bits long.

If we want to correct single-bit errors, how many parity bits do we need?
How many parity bits?

- If we want each of the $n$ bits of the codeword to belong to a unique subset of the $n-k$ parity bits, then

$$n \leq 2^{n-k} - 1$$

recalling that a set with $s$ elements has $2^s - 1$ non-empty subsets.

- The $n-k$ parity bits are assigned to the $n-k$ singleton sets, each message bit is assigned to unique subset of the parity bits (i.e., each message bit is covered by a unique combination of two or more parity bits).

- Most efficient when $n$ exactly matches the number of non-empty subsets of parity bits: $n = 2^{n-k} - 1$
  - $(7,4)$, $(15,11)$, $(31, 26)$ Hamming codes

This code is shown on slide 10
What \((n,k)\) code does one use?

- The minimum Hamming distance \(d\) between codewords determines how we can use code:
  - To detect \(D\)-bit errors: \(d > D\)
  - To detect and correct \(C\)-bit errors: \(d > 2C\)
  - To detect \(D\)-bit errors and correct \(C\)-bit errors (for \(D > C\)): \(d > D + C\)
  - Sometimes code names include min Hamming distance: \((n,k,d)\)

- To conserve bandwidth want to maximize a code’s \textit{code rate}, defined as \(k/n\).

- Parity is a \((n+1,n,2)\) code
  - Efficient, but only 1-bit error detection

- Replicating each bit \(r\) times is a \((r,1,r)\) code
  - Simple way to get great error correction, but inefficient
A simple (8,4,3) code

Idea: start with rectangular array of data bits, add parity checks for each row and column. Single-bit error in data will show up as parity errors in a particular row and column, pinpointing the bit that has the error.

Parity for each row and column is correct ⇒ no errors

Parity check fails for row #2 and column #2 ⇒ bit D4 is incorrect

Parity check only fails for row #2 ⇒ bit P2 is incorrect

If we add an overall parity bit P5, we get a (9,4,4) code!
Correcting single-bit errors is nice, but in many situations errors come in bursts many bits long (e.g., damage to storage media, burst of interference on wireless channel, ...). How does single-bit error correction help with that?

Well, can we think of a way to turn a B-bit error burst into B single-bit errors?

Problem: Bits from a particular codeword are transmitted sequentially, so a B-bit burst produces multi-bit errors.

Solution: **interleave bits** from B different codewords. Now a B-bit burst produces 1-bit errors in B different codewords.
Framing

• Looking at a received bit stream, how do we know where a block of interleaved codewords begins?

• Physical indication (transmitter turns on, beginning of disk sector, separate control channel)

• Place a unique bit pattern (frame sync sequence) in the bit stream to mark start of a block
  - Frame = sync pattern + interleaved code word block
  - Search for sync pattern in bit stream to find start of frame
  - Bit pattern can’t appear elsewhere in frame (otherwise our search will get confused), so have to make sure no legal combination of codeword bits can accidentally generate the sync pattern (can be tricky…)
  - Sync pattern can’t be protected by ECC, so errors may cause us to lose a frame every now and then, a problem that will need to be addressed at some higher level of the communication protocol.
Summary: example channel coding steps

1. Break message stream into k-bit blocks.

2. Add redundant info in the form of (n-k) parity bits to form n-bit codeword. Goal: choose parity bits so we can correct single-bit errors, detect double-bit errors.

3. Interleave bits from a group of B codewords to protect against B-bit burst errors.

4. Add unique pattern of bits to start of each interleaved codeword block so receiver can tell how to extract blocks from received bitstream.

5. Send new (longer) bitstream to transmitter.

Sync pattern has five consecutive 1’s. To prevent sync from appearing in message, “bit-stuff” 0’s after any sequence of four 1’s in the message. This step is easily reversed at receiver (just remove 0 after any sequence of four consecutive 1’s in the message).
Summary: example error correction steps

1. Search through received bit stream for sync pattern, extract interleaved codeword block
2. De-interleave the bits to form B n-bit codewords
3. Check parity bits in each code word to see if an error has occurred. If there's a single-bit error, correct it.
4. Extract k message bits from each corrected codeword and concatenate to form message stream.
Summary

- To detect D-bit errors: Hamming distance > D
- To correct C-bit errors: Hamming distance > 2C
- To detect D-bit errors and correct C-bit errors (for D > C): Hamming distance > C + D
- \((n,k,d)\) codes have code rate of \(k/n\)
- For our purposes, we want to correct single-bit errors and detect double bit errors, so \(d = 4\)
- Handle B-bit burst errors by interleaving B codewords
- Add sync pattern to interleaved codeword block so receiver can find start of block, bit stuff message to make sync unique.
- Use checksum/CRC to detect uncorrected errors in message