

C:\cygwin\home\zahorjan\cse461\12au\04-errors.cp3

CSE 461

Errors: Detection & Correction

How Hard Should we Try to Fix Link Errors?

The “End to End Argument” (1984):

- *Functionality should be implemented at a lower layer only if it can be correctly and completely implemented. (Sometimes an incomplete implementation can be useful as a performance optimization.)*

So:

- Keep residual error rate low, but don't aim for perfection

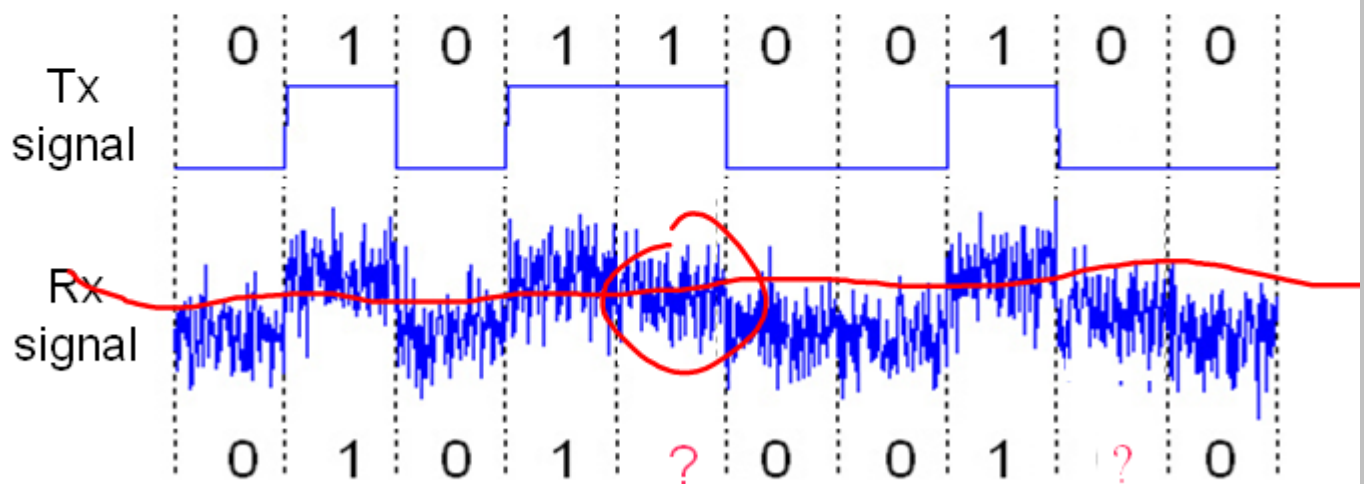
In practice:

- Use error detection on fiber, wires (Why?)
- Add retransmissions to wireless links (Why?)

CNSE by Tanenbaum & Wetherall, © Pearson Education—Prentice Hall and D. Wetherall, 2011

Errors -- Problem

- Noise in the received signal can flip the decoded bits
 - We must be able to recover when this occurs!



Dealing with Errors: Approaches

- What should be done about errors?
 - It depends...
- You might do nothing
 - When?
- You might re-send until you're sure the data has arrived
 - When?
 - *Automatic Repeat reQuest* (ARQ)
 - Requires *error detection*
- You might pre-send (something like) multiple copies
 - When?
 - *Forward Error Correction* (FEC)
 - Requires *error correction*

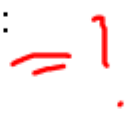
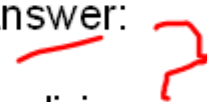
Sender
re-send

) pre-send

Dealing with Errors: Redundancy

- Redundancy is “extra bits sent”
 - They’re extra because they don’t increase the information content (entropy) of the message
- Redundancy is essential to:
 - **Error detection**: codes allow errors to be recognized by receiver
 - **Error correction**: codes allow errors to be repaired by receiver

Dealing with Errors: Redundancy

- Simple, motivating error detection scheme:
 - Just send two copies of each bit:
 - 10100011 10100011 
 - 11001100 00001111
 - What can receiver do?
 - Detect errors?
 - Correct errors?
- Coding question: Can we do any better? Answer: 
 - Yes – stronger protection with fewer bits!
 - But we can't catch all inadvertent errors, nor malicious ones
- This module is about coding for detection and correction
 - Retransmission is the next module

Codes for Error Detection/Correction

- We will look at basic block codes
 - K bits in, N bits out is a (N,K) code; a memoryless mapping
- K message data bits N-K check bits

check bits = F(data bits)
- Implies we use only 2^K distinct bit strings of the 2^N possible.
 - Sender computes the N-K check bits based on the data
 - Receiver also computes check bits based on the data it receives
 - Mismatches mean...?
 - Mapping to the “closest” valid codeword can correct errors
 - so long as not too many bits are corrupted.
 - Example: sending words ‘cow’, ‘pig’, and ‘horse’
 - What was sent if I receive ‘cog’?

Evaluating Codes

Block codes are characterized in two ways:

- Overhead, or rate (fraction) of useful information
 - Ex: 1000 data bits + 100 check bits = rate of 10/11, 10% overhead
- The errors they detect/correct
 - For example:
 - all single-bit errors
 - all errors of no more than 3 bits
 - all bursts of no more than 3 bits, etc.

Q: How can we catch many/likely errors with low overhead?

A: There are many specific coding schemes. The book covers the details of some of them.

The Hamming Distance

- Gives us bounds on errors that can reliably be detected
- Hamming distance of a code is the smallest number of bit differences that turn any one codeword into another
 - Example: code is 000 for 0, 111 for 1, then...
 - Hamming distance is 3
- For code with distance $d+1$:
 - d errors can be detected (e.g, 001, 010, 110, 101, 011)
- For code with distance $2d+1$:
 - d errors can be corrected (e.g., 000, 111)

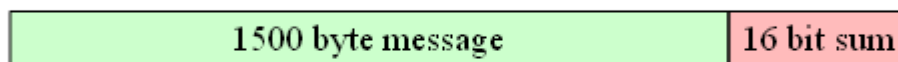
Simplest Error Detection Code – Parity

- Start with k bits and add another so that the total number of 1s is even (*even parity*)
 - e.g. 0110010 \rightarrow 0110010 1
 - Easy to compute as XOR of all input bits

- Will detect 1 bit \rightarrow odd # of bits
- Will not detect even # errors
- Corrects nothing
 - Unless the errors are erasures

Checksums

- Used in Internet protocols (IP, ICMP, TCP, UDP)
- Basic Idea: Add up the data and send it along with sum



- Algorithm:
 - Mouthful for “sum”: “checksum is the 1s complement of the 1s complement sum of the data interpreted 16 bits at a time” (for 16-bit TCP/UDP checksum)
 - 1s complement nit: flip all bits to make a number negative, so adding requires carryout to be added back.
- Will detect _____.

Internet checksum properties

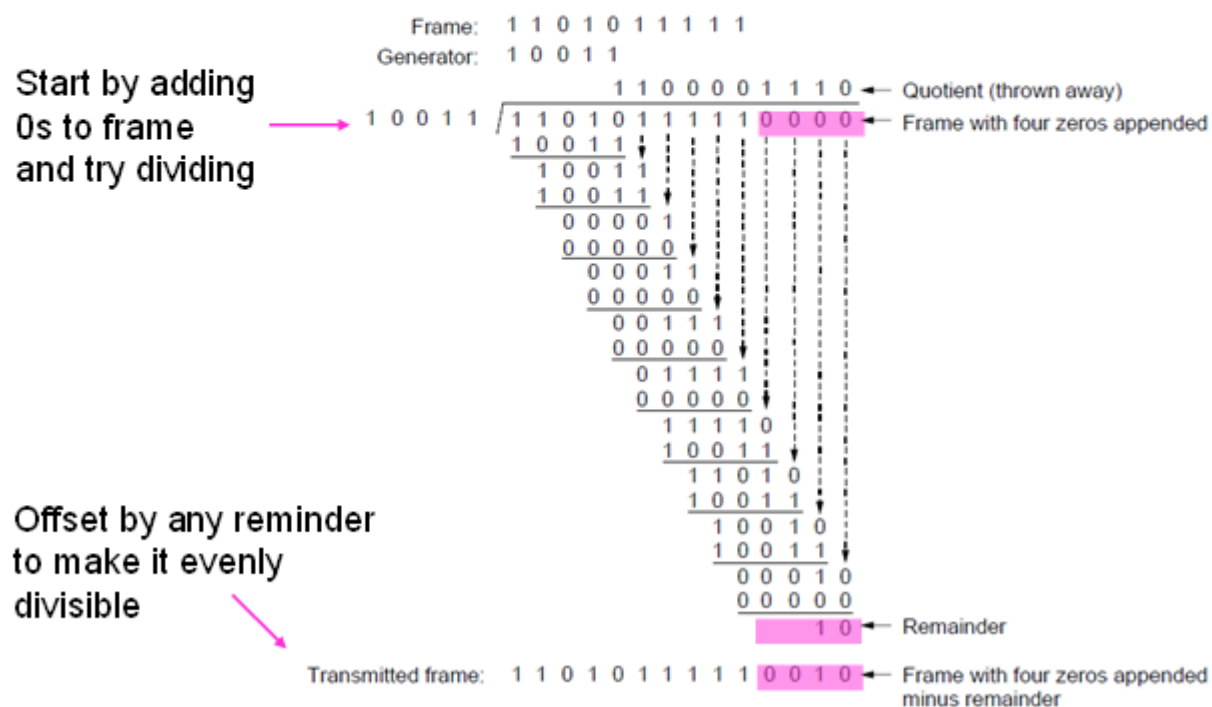
- Catches all error bursts < 16 bits
- Random errors detected with prob. $1 - 2^{-16}$
- Fails to catch transpositions, insertion/deletion of zeros
 - These are typically hardware/software bugs not random errors

CRCs (Cyclic Redundancy Check)

- Stronger protection than checksums
 - Used widely in practice, e.g., Ethernet/802.11 CRC-32
- Algorithm: Given n bits of data, generate a k -bit check sequence that gives a combined $n + k$ bits that are divisible by a chosen divisor $C(x)$
- Based on mathematics of finite fields
 - “numbers” correspond to polynomials, use modulo arithmetic
 - e.g, interpret 10011010 as $x^7 + x^4 + x^3 + x^1$

CRC Example

Adds bits so that transmitted frame viewed as a polynomial is evenly divisible by a generator polynomial



CN5E by Tanenbaum & Wetherall, © Pearson Education-Prenice Hall and D. Wetherall, 2011

“Standard” CRC-32

- It is:
 - CRC-32: 10000010011000001000111011011011
 - Used for Ethernet, cable modems, ADSL, PPP, ...
- Q: What kind of errors will/won't checksums detect?
 - All 1 and 2 bit errors
 - All burst errors < 32 bits
 - All errors with an odd number of flips
 - All based on mathematical properties; look in the book
 - Random errors with prob $1 - 2^{-32}$
- Stronger than checksums

DVD Error Detection

16.4 Error Detection Code (EDC)

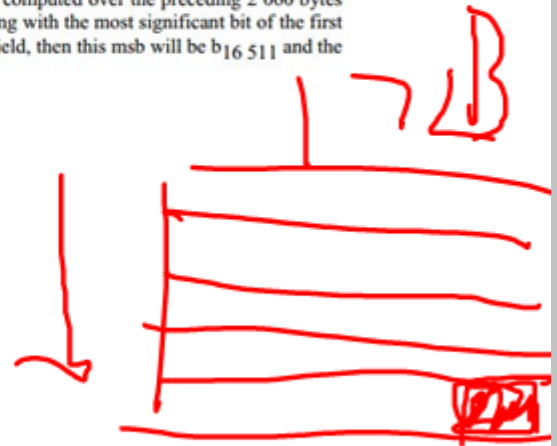
This 4-byte field shall contain the check bits of an Error Detection Code computed over the preceding 2 060 bytes of the Data Frame. Considering the Data Frame as a single bit field starting with the most significant bit of the first byte of the ID field and ending with the least significant bit of the EDC field, then this msb will be $b_{16\ 511}$ and the lsb will be b_0 . Each bit b_i of the EDC is as follows for $i = 31$ to 0 :

$$\text{EDC}(x) = \sum_{i=31}^0 b_i x^i = I(x) \bmod G(x)$$

where

$$I(x) = \sum_{i=16\ 511}^{32} b_i x^i$$

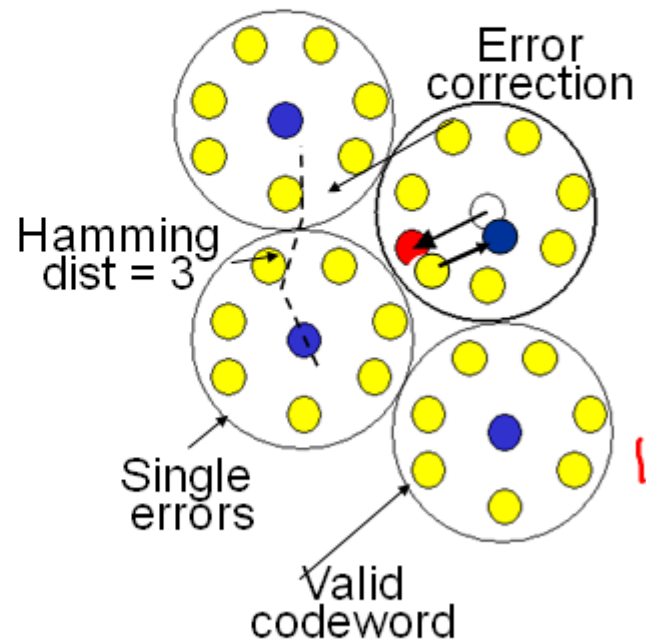
$$G(x) = x^{32} + x^{31} + x^4 + 1$$



<http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-267.pdf>

Error correcting codes

- Errors create clusters around codewords
- Correct by mapping to nearest codeword
- Works for d errors if distance $\leq 2d+1$



Error Correcting Code Example

- (3, 1) repetition code has codewords 000 and 111
- Decoding:
 - 000 \rightarrow 0 (no error)
 - 001 \rightarrow 0
 - 010 \rightarrow 0
 - 011 \rightarrow 1
 - 100 \rightarrow 0
 - 101 \rightarrow 1
 - 110 \rightarrow 1
 - 111 \rightarrow 1 (no error)
- Corrects up to 1 error; better not have a double error!
- Other error-correcting codes are more complicated

Patterns of Errors Matter

- Q: Suppose you expect a bit error rate (BER) of about 1 bit per 1000 sent. What fraction of packets would be corrupted if they were 1000 bits long (and you could detect all errors but correct none)?

Patterns of Errors

- A: It depends on the pattern of errors
 - Bit errors occur at random
 - Packet error rate is about $1 - 0.999^{1000} = 63\%$
 - Errors occur in bursts, e.g., 100 consecutive bits every 100K bits
 - Packet error rate $\leq 2\%$

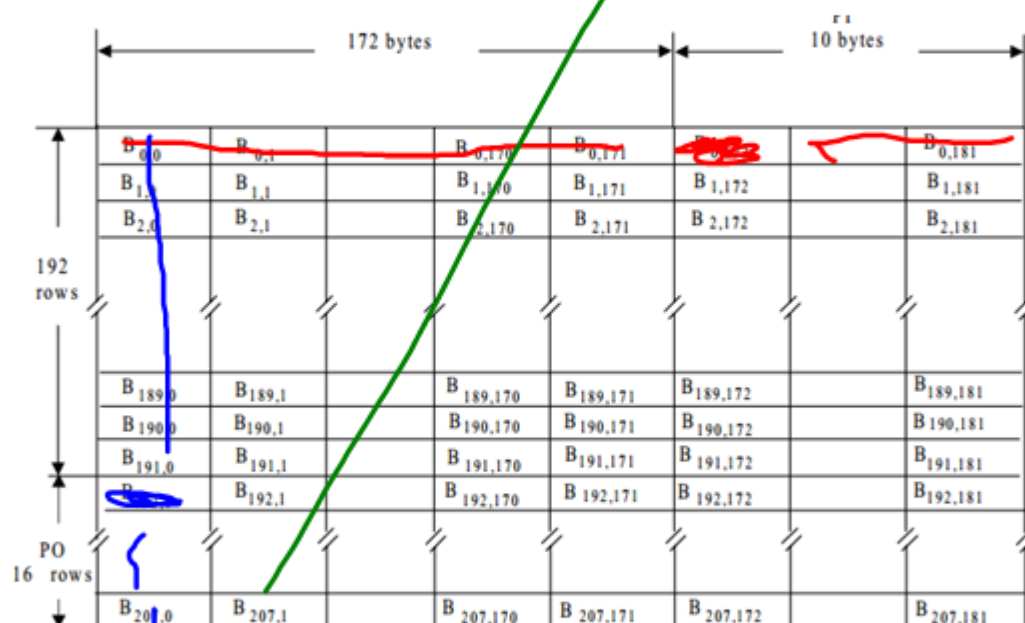
Real Error Models

- Random, e.g., thermal noise as in AWGN
- Bursty, e.g., wires, if there is an error it is likely to be a burst
 - Common due to physical effects
- Errors can also be “erasures”, e.g., lost packet
- Q: Which of the above are most/least easy to handle?

Bursty Errors

- Common in practice, but difficult to handle
 - “some messages are a lot wrong” vs “most message are only a little wrong”
- Two strategies:
 - A code designed to handle them well,
 - E.g., CRCs, Reed-Solomon codes
 - Use interleaving to convert them to random errors and use codes that are good for random errors

DVD Error Correction



SP-48022-A

Figure 20 - ECC Block

<http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-267.pdf>

Interleaving – a neat trick

- Compute check (parity) bit across items, not per item
 - Tolerates burst errors, at the cost of added latency

