Error Correction and Detection - Outline

- Introduction
- Types of Error Detection and Correction
- Hamming Codes
- Basic Checksums

Introduction

- Michelle Lorenz
  - Michelle@Lorenz.co.nz
- Course Coverage:
  - Error Detection
  - Error Correction
  - Data Compression
  - Ethernet Protocols

Error Detection and Control

- Data transmission is traditionally error prone, but computers really prefer data to be completely error-free
- There are two ways of handling transmission errors
  - Error detection and correction
  - Error detection and retransmission

Error Correction

- Often called Forward Error Control (FEC)
- Includes extra, redundant information so that lost data can be reconstructed
- FEC is expensive, especially to handle burst errors rather than occasional bit errors
- FEC must be used where original data is not available, such as deep-space telemetry and data recording

Error Detection and Retry

- Also known as Automatic Repeat Request (ARQ)
- Relies on error detection and retransmission of faulty messages
- The usual method in data communications
- Very good error detection and much simpler than moderate error correction
- Uses ‘checksums’ to only *detect* errors, not *correct* them

FEC and ARQ ctd

- For both FEC and ARQ, we calculate a checksum at the sender and transmit this with the data
- At the receiver the checksum is recomputed and checked for agreement with what is transmitted

FEC

- Hamming Codes are an example of FEC
- Hamming Codes insert parity bits into the message
- Parity is the extra bit which is added to the information bits and adjusted so that the overall bit count is either odd or even

Parity

- Odd parity: 00101010p_1 00101110p_2
  - p_1=0, p_2=1
- Even parity: 00101010p_1 00101110 p_2
  - p_1=1, p_2=0
- All error correction uses parity bits, usually many of them, each checking some of the bits so that the pattern of bits where parity fails indicates the position of the error
- Parity is often described using:

A message of content and clarity,
Has gotten to be quite a rarity –
To combat the terror
Of serious error,
Use bits of appropriate parity!

How many parity bits do we need?

- We have an 8-bit message
- If we use 1 parity bit, the message either fails or succeeds
- If we break the message into 2 and use 2 parity bits on the message, one of four conditions may occur
- Typically for *n* parity checks there are 2^n possible combinations for success or failure

Forward Error Correction – Hamming Codes

- With a message of $k = 2^n - 1$ bits, proper coding of an n-bit error field should be able to indicate one of 2^n *conditions:*
  - *No error at all, OR*
  - *which of the $k = 2^n - 1$ bits has a single error*

Single Error Correcting Codes (SEC)

- With $k = 2^n - 1$ bits in the codeword and n bits used for checking, there are $i = k - n$ bits of available information
- This leads to the family of possible Single Error Correcting (SEC) codes, described by their (codeword_length, data_length)

SEC

n	*k*	*i*	Description	
2	3	1	(3,1)	A 2-out-of-3 majority code
3	7	4	(7,4)	The 'usual' Hamming Code We use a version of this one
4	15	11	(15,11)	
5	31	26	(31,26)	
6	63	57	(63,57)	

- There are many codes that fit this specification, but the simplest and oldest is the Hamming Code

SEC ctd.

- To handle 8-bit data, we must use the (15,11) code, discarding 3 data bits to get a (12,8) code
  - Take a 12-bit 'word' with bits numbered 1…12
  - Allocate bits $2^{0..n-1}$ to parity bits (1,2,4,8) leaving the rest to data
  - Generate parities according to the the table:

SEC ctd.

- Assume an 8-bit message:
  - m1 m2 m3 m4 m5 m6 m7 m8
- And 4 parity bits p1 p2 p3 p4

	1	2	3	4	5	6	7	8	9	10	11	12
	p1	*p2*	*m1*	*p3*	*m2*	*m3*	*m4*	*p4*	*m5*	*m6*	*m7*	*m8*
Group 1	X		X		X		X		X		X	
Group 2		X	X			X	X			X	X	
Group 3				X	X	X	X					X
Group 4								X	X	X	X	X

- Parity 1 checks all positions whose 1st lsd = 1
- Parity 2 checks all positions whose 2nd lsd = 1

Example

- The 8-bit information word 01001010 expands first to *pp*0*p*100*p*1010, creating space for the parity bits
- Assume odd parity (which ensures that there is at least 1 1-bit)

	1	2	3	4	5	6	7	8	9	10	11	12
	p1	*p2*	*0*	*p3*	*1*	*0*	*0*	*p4*	*1*	*0*	*1*	*0*
Group 1	X		X		X		X		X		X	
Group 2		X	X			X	X			X	X	
Group 3				X	X	X	X					X
Group 4								X	X	X	X	X

Example Ctd.

	1	2	3	4	5	6	7	8	9	10	11	12
	p1	*p2*	*0*	*p3*	*1*	*0*	*0*	*p4*	*1*	*0*	*1*	*0*
Group 1	X		X		X		X		X		X	
Group 2		X	X			X	X			X	X	
Group 3				X	X	X	X					X
Group 4								X	X	X	X	X

Then,	*p1* checks *p1, m1, m2, m4, m5, m7*	0,1,0,1,1	*p1* = 0
	p2 checks *p2, m1, m3, m4, m6, m7*	0,0,0,0,1	*p2* = 0
	p3 checks *p3, m2, m3, m4, m8*	1,0,0,0	*p3* = 0
	p4 checks *p4, m5, m6, m7, m8*	1,0,1,0	*p4* = 1

- The final code word is 000010011010 (parity bits underlined)

Checking for an error

- We send the message: 000010011010
- But get: 000011011010
- Without error correction we extract the message: 01101010
- Which is not equal to: 01001010
- We need to count the 1s at each mask
  - If the count is odd, generate a syndrome bit=0
  - If the count is even, generate a syndrome bit=1

Correction from Lecture Handouts

	0	*0*	*0*	*0*	*1*	*1*	*0*	*1*	*1*	*0*	*1*	*0*	*Count*	*Syn-drome*
Group 1	X		X		X		X		X		X		3	0
Group 2		X	X			X	X			X	X		2	1
Group 3				X	X	X	X					X	2	1
Group 4								X	X	X	X	X	3	0

- Groups 2 and **3** fail; the error bit is in **0110** = 6
- Correct the received data to: 000010011010
- Delete the parity bits: 01001010
- Which is the same as the original: 01001010

Error Correction and Detection

- The number locating the error is known as the *syndrome*, and is usually zero if no error
- We can add a single overall parity bit to get a SECDED code (Single Error Correcting, Double Error Detecting)
- If two bits are corrupted, the overall parity is still OK, but at least one internal parity fails, signaling an uncorrectable error

FEC Ctd.

- Codes which correct multiple errors are much more difficult
- Compact discs have very powerful error correction
  - BER is the bit error rate in reading from the CD
    - BER = the percentage of bits that have errors
  - The maximum correctable burst is 4000 data bits
  - Uncorrected errors = < 1 in 750 hours = BER 10^{-3}
  - Undetectable errors; BER = 10^{-4}

Error Correction with ARQ

- Data communications errors tend to be rare and occur in bursts
- Both aspects make it very difficult to design good error correction codes
- It is more efficient to use very powerful error detection, with retransmission (Automatic Repeat Request – ARQ)

ARQ

- None of these checksums (1s complement, Fletcher, Alder or CRC is *designed* to correct errors (although some can do that to a very limited extent as in ATM cell headers); they are designed as error *detectors*. (The Hamming is a Single-Error-Correcting code [SEC]; with an extra parity bit it becomes a Double-Error-Detecting code as well [SEC-DED], but no more)
- All are designed to give very good error detection (undetected errors often less than 1 in 10^9)
- The checksums all cover entire messages
- Generally we ignore character parity from now on

Additive Checksums

- These codes work by performing *arithmetic addition* on the words of the message
- The simplest is the TCP/IP checksum, which is the 1s complement sum of all of the (16 bit) words of the message – assume 32-bit arithmetic

TCP/IP Checksum

```
sum += word;                  //do the addition
while (sum > 0xFFFF)          //if beyond 16 bits
      sum = (sum & 0xFFFF)    //isolate 16 low bits
            + (sum >> 16)     //and add the 'overflow'
```

- It is simple but not very good
  - An error in one bit has a rather local effect on the checksum
  - All words are treated equally; it is insensitive to transpositions
- Better checksums give different weights to different message positions

Recommended Reading

- Understanding Data Communications and Networks
  - Pages 241 – 244 (Hamming Codes – Single bit error correction)