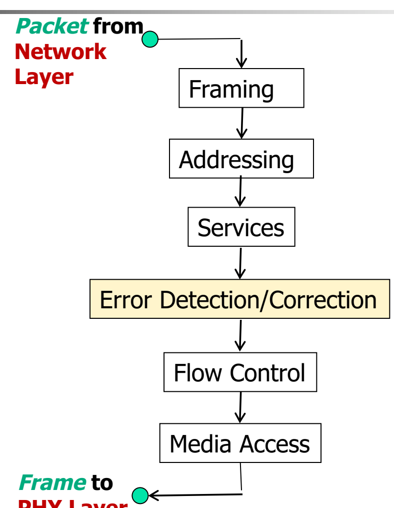
The Data Link Layer – Error Control

CSC 570 Computer Networks

Fall 2019

DLL Block Diagram



Bit Errors

- Bit errors can happen over communication channels due to channel noise and interference
  - Transmitted 1 decoded as 0 at receiver (or 0 taken as 1)
- Can happen at *random* bits or in *bursts*

Khaled Harfoush 2019

Error Control

- Bit errors can be treated using one of two ways:

1. *Error detection* techniques, ACKs, and retransmissions in case of errors
   - Timers are used to protect against lost ACKs/frames
2. *Error correction* techniques, with NO need for ACKs and retransmissions – but still more redundancy (overhead) needed

Khaled Harfoush 2019

Detection vs Correction

- When channels are *lossy* by nature (e.g. wireless links, satellite links), it is more beneficial to do error correction -- costly
- When channels are *not* lossy (e.g. fiber), it is enough to do error detection – less costly
- But how? The idea is to add *redundant* bits

Terminology

- *Redundancy* is added to data so errors can be either detected, or corrected.
- Consider a *frame* of *m* data bits, and *redundancy* of *r* redundant (check) bits

  Let $n \equiv m+r$

  We describe this code as (n,m)

  Code rate $\equiv m/n$

  e.g. code rate=1/2 for noisy channel

  code rate≈1 for a high quality channel

Theory

- Number of possible data messages = 2^m
- Number of *legal codewords* (of size $n=m+r$) is still 2^m
- So not all 2^n codewords are legal
- Fraction of legal codewords=$2^m/2^{m+r}=1/2^r$

- It is the *sparseness* with which the message is embedded in the space of codewords that allows the receiver to *detect* and *correct* errors

Hamming Distance

- The Hamming distance is the *minimum* number of bit flips to turn one *valid codeword* into any other *valid* one.

- Example:

  With 4 codewords of 10 bits:

0000000000, 0000011111, 1111100000, and 1111111111

  Hamming distance is 5

Detection and Correction Properties

1. Legal Codewords with Hamming distance of *d+1* will *detect* up to *d* bit errors

2. Legal codewords with Hamming distance of *2d+1* will *correct* up to *d* bit errors

Why Can a Code with Distance *d+1* Detect up to d Errors?

- Because errors are detected by receiving invalid codewords
- If there are *d+1* or more errors then one valid codeword may be turned into another valid codeword and there is no way to detect that an error has occurred.
  - Example: sending 0000000000 with 4 flips might give 1111000000 which is invalid, detecting the error. But with 5 flips 1111100000 might be received, which is a another valid but not the one transmitted, which is still an error, but it cannot be detected.

Why Can a Code with Distance *2d+1* Correct up to d Errors?

- Because errors are corrected by mapping a received invalid codeword to the nearest valid codeword, i.e., the one that can be reached with the fewest bit flips.
- If there are more than *d* bit flips, then the received codeword may be closer to another valid codeword than the one that was sent.
  - Example: sending 0000000000 with 2 flips might give 1100000000 which is closest to 0000000000, correcting the error. But with 3 flips 1110000000 might be received, which is closest to 1111100000, which is still an error.

Q:

- With 4 codewords of 10 bits:

0000000000, 0000011111, 1111100000, and 1111111111

Hamming distance is 5.

1. How may bit errors can be detected? 4
2. How many bit errors can be corrected? 2
3. Can you both detect and correct these many errors? No. Why?

A:

- When correcting up-to 2 bit errors, we map the received codeword to the closest legal codeword. Example:
  - Received 1111000000 would map to 1111100000
- When detecting up-to 4 bit errors, errors will be detected by possibly due to
  - 1111100000 being delivered as 1111000000, or
  - 0000000000 being delivered as 1111000000
- Doing both detection and correction would require us to interpret a received codeword in 2 different ways

Q:

- Design an (n,m) code, where $n \equiv m+r$, capable of *correcting* all *single errors*.
- What is the corresponding *minimum* value of r?

A (1/2):

- There are 2^m *legal codewords* of size $n=m+r$
- In order to *correct* single errors, each *legal codeword*, C, needs *n illegal codewords* around it – not shared with illegal codewords associated with other legal codewords. These will be mapped to C when received
- These can be attained by systematically flipping the *n* bits of C, one at a time.

A (2/2):

- Thus each of the 2^m legal codes will need *n+1* points in the codewords of n-dimensional space
- So we must have

  $(n+1)2^m \leq 2^n$

  Using $n=m+r$

  $(m+r+1) \leq 2^r$

Terminology

1. *Block codes:* *m* and *r* have fixed sizes, and *r* is computed solely based on the corresponding *m* bits.
2. *Systematic codes:* The input *m* is also used as the output data, rather than being encoded before being sent.
3. *Linear codes:* The *r* check bits are computed as a linear function of the *m* data bits
   - Popular linear functions: *XOR* and *modulo-2 addition*.

Agenda

1. Error Correction Codes
2. Error Detection Codes

Error Correction codes

1. Hamming codes
2. Binary convolutional codes
3. Reed-Solomon
4. Low-Density Parity Check codes

1. Hamming Codes

- *Systematic, linear block code*: Provides check bits to correct up to a single bit error
- The bits of the codeword are numbered from left to right starting with position 1.
- The check bits occupy power of 2 positions (1,2,4,8, etc)
- The message bits occupy the other positions (3,5,6,7, etc)

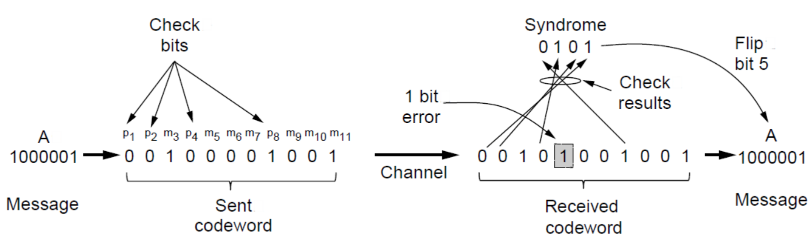
Generating Hamming Codes

- Check bits are parity (even or odd) over subsets of the codeword
  - A data bit may be included in several check bit computations
  - The data bit at position k contributes to the check bits in positions consisting of the power of 2 numbers summing up to k.

- Example:

Data bit in position 11 contribute to check bits in positions 1,2,8 since 11=1+2+8

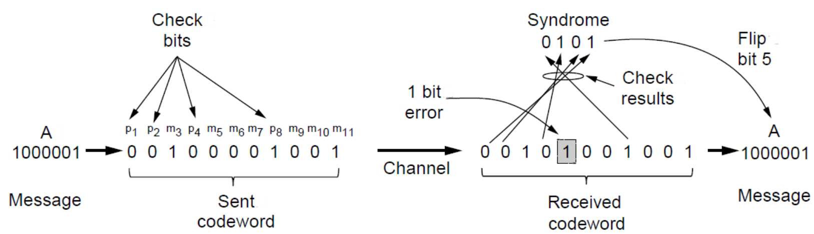
Example



adds 4 check bits and can correct 1 error

Validating Hamming Codes

- Re-computing the parity sums (*error syndrome*) gives the position of the error to flip, or 0 if there is no error



adds 4 check bits and can correct 1 error

2. Convolutional Codes

- *Non-block code*: Operates on a stream of bits, keeping internal state
- Output stream is a function of preceding input bits – Encoder has *memory*
- Generates parity symbols via the sliding application of a boolean polynomial function to the data stream
- Popular NASA binary convolutional code used in 802.11, GSM, Satellite communications

Terminology

- $n \equiv$ input data rate
  $k \equiv$ output symbol rate
  base code rate = n/k
  $v \equiv$ The depth (number of memory elements) -- *constraint length*
  $2^v \equiv$ Number of states
- Convolutional codes are often characterized by $[n,k,v]$.
- The output is a function of (1) the current input as well as (2) the previous $v-1$ inputs.

Convolutional Encoding

1. Start with *v memory registers*, each holding one input bit (starting with 0 values)
2. The encoder has *k* modulo-2 adders and *k* generator polynomials — one for each adder
3. An input bit m_1 is fed into the leftmost register. Using the generator polynomials and the existing values in the remaining registers, the encoder outputs *k* symbols
4. Now bit *shift* all register values to the *right*

Example

- rate $^{1}/_{3}$ encoder with 8 states. Generator polynomials are $G_1 = (1,1,1), G_2 = (0,1,1)$, and $G_3 = (1,0,1)$.

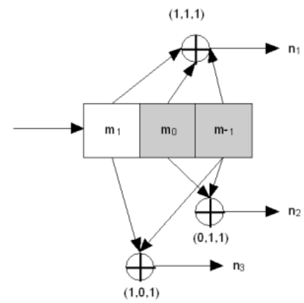
  Therefore, output bits are calculated (modulo 2) as follows:

  $n_1 = m_1 + m_0 + m_{-1}$

  $n_2 = m_0 + m_{-1}$

  $n_3 = m_1 + m_{-1}$

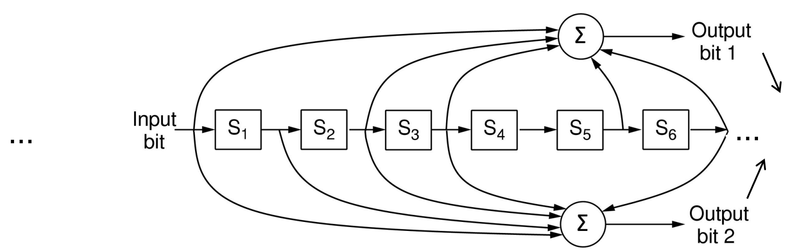
- Encoder in this example is *linear* and *non-systematic*



Rate 1/3 convolutional encoder with constraint length 3

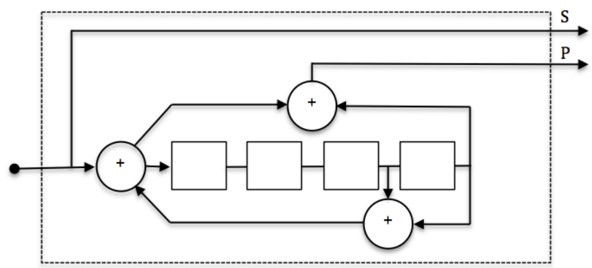
Other Examples (1/2)

- *64 states, ½ rate, linear, non-block, non-systematic* convolutional code



Other Examples (2/2)

- *16 states, ½ rate, linear, non-block, systematic* convolutional code



Convolutional Decoding

- Bits are decoded with the *Viterbi algorithm*
- Returns the bits that form the most likely codeword
- Convolutional codes are effective at dealing with *isolated errors*, but will likely fail with *bursts of errors*

3. Reed-Solomon Code

- *Systematic, linear block code*
- Unlike Hamming codes, which operate on individual bits, Reed-Solomon codes operate on *m*-bit symbols
  - A single bit error and an *m*-bit burst error are both treated simply as *one symbol error*
- Widely used for DSL, cable, satellite communication, DVDs, Blue-ray discs because of the strong error correction properties especially for *burst errors*

Idea

- Every *n* degree polynomial is uniquely determined by *n+1* points. Extra "check" points on the same line are redundant and are useful for error correction
- If a point is received in error, we can still recover the data points by fitting a line to the received points.

Details

- Let *m=8*
- A *symbol* is *m* bits long
- A *codeword* is 2^8-1=255 bytes long
- The *(255,233)* code adds *32 redundant symbols* to *233 data symbols*.
- When *2t redundant symbols* are added, a Reed-Solomon code is able to *correct* up to *t symbol errors*.
  - The *(255,233)* code can thus correct up to 16 symbol errors (16.8=128 bits)

Khaled Harfoush 2019

Deployment

- *Reed-Solomon* codes are effective at dealing with *bursts of errors*
- Often used *in combination with convolutional codes* – which is good at correcting *isolated errors*

Khaled Harfoush 2019

4. Low-Density Parity Check Code

- *Systematic, linear block code*
- Practical for large block sizes
- Used in 10 Gbps Ethernet, power line networks, latest versions of 802.11

Khaled Harfoush 2019

Idea

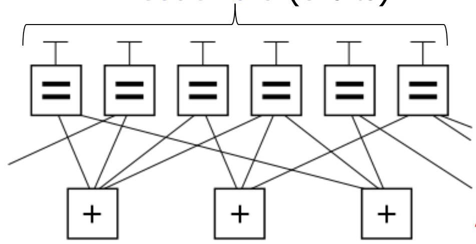
- Leads to a matrix representation of the code that has low density of 1s, hence the name
- The received codewords are decoded with an approximation algorithm that iteratively improves on a best fit of the received data to a legal codeword. This corrects errors

Khaled Harfoush 2019

Graphical Representation of (n,k) LDPC Code

(6,3) code – 3 bit messages encoded as 6 bits

Codeword (6 bits)



n variable nodes

n-k constraint nodes

- Lines out of a variable node have *same values*
- For a legal codeword, lines into each constraint node must *sum up to 0 (mod 2)*

Example (1/3)

- In the previous (6,3) code

1. there are eight possible six-bit strings corresponding to valid codewords: (i.e., 000000, 011001, 110010, 101011, 111100, 100101, 001110, 010111)
2. the *parity-check matrix* representing this graph is

Each row represents one of the 3 constraints

$$H=\begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

Each col represents one of the 6 bits of a codeword

Example (2/3)

3. The parity check matrix, H, is converted into a *Generator* matrix, G, which when multiplied by a message (of 3-bits in this case), generates the corresponding legal codeword (of 6-bits)

$$G = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 \end{bmatrix}$$

Example (3/3)

[1 0 1] . G = [1 0 1 0 1 1]

101011 is a legal codeword

message

redundancy

- G is used to generate codewords (adding redundancy), and is used at the receiver to validate codewords

Agenda

1. Error Correction Codes
2. Error Detection Codes

Error Detection codes

1. Parity
2. Checksums
3. Cyclic redundancy codes

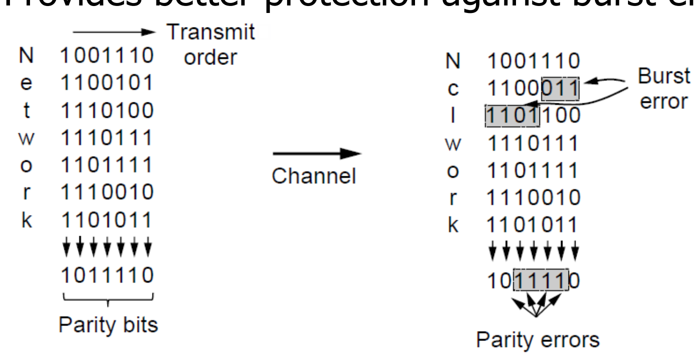
1. Parity

- Parity bit is added as the *modulo 2 sum* of data bits
  - Equivalent to *XOR*; this is even parity
  - Ex: 1110000 → 11100001
  - Detection checks if the sum is wrong (an error)
- Simple way to *detect* an *odd* number of errors
  - Ex: 1 error, 11100101; detected, sum is wrong
  - Ex: 3 errors, 11011001; detected sum is wrong
  - Ex: 2 errors, 1110110 ; *not detected*, sum is right!
  - Error can also be in the parity bit itself

Khaled Harfoush 2019

Interleaving Parity Bits

- Each parity sum is made over non-adjacent bits
- Provides better protection against burst errors



Khaled Harfoush 2019

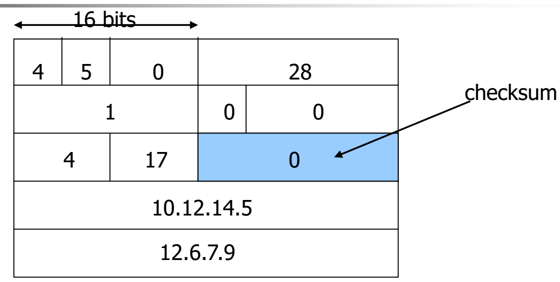
Idea

- A burst error will be spread across multiple rows, and errors will be checked by different parity bits
- *Interleaving* of N parity bits *detects burst errors* up to N

2. Checksums

- Checksum treats data as *N-bit words* and adds *N check bits* that are the *modulo 2^N sum of the words*
- Properties:
  - Improved error detection over parity bits
  - Detects bursts up to N errors
  - Detects random errors with probability $1\text{-}2^N$
  - Vulnerable to systematic errors, e.g., added zeros
- Ex: Internet 16-bit 1s complement checksum

Sender Checksum calculation



1. Divide the portion of the packet for which the checksum is computed (header) into *16-bit words* (the checksum field included is filled with 0s)
2. All sections are added together using *one's complement addition*
3. The final result is *complemented* to make the final checksum

Sender Checksum calculation

4,5, and 0 → 01000101 00000000
28 → 00000000 00011100
.
.
.
7, 9 → 00000111 00001001

Sum (16-bit only after adding carry-ons) → 01110100 01001110
Checksum → *10001011* *10110001*

Receiver Checksum calculation

- The receiver does the same checksum calculation (but now the checksum field $\neq$ 0s)
- If the result $\neq$ 0s
  $\rightarrow$ the packet is corrupted and dropped

3. CRC

- Also known as *polynomial code*
- Treats bit strings as polynomials with coefficients of 0 and 1
  - Example: 1101 maps to $1.x^3+1.x^2+0.x^1+1.x^0=x^3+x^2+1$
- Sender and receiver agree on a generator polynomial, G(x)
- Adds bits so that transmitted frame viewed as a polynomial is evenly *divisible by* G(x)
- Stronger detection than checksums
  - Not vulnerable to systematic errors
- Ethernet 32-bit CRC is defined by:

$$x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^{8}+x^{7}+x^{5}+x^{4}+x^{2}+x^{1}+1$$

Polynomial Arithmetic

- Modulo 2 arithmetic
- *Addition* and *subtraction* identical to XOR
- *Division* is identical to binary division modulo 2

CRC Sender Algorithm

1. Let r be the degree of $G(x)$. Append r zero bits to the end of the frame $M(x)$ of m bits. Result has $m+r$ bits and corresponds to a polynomial of $x^r M(x)$
2. Divide $x^r M(x)$ by $G(x)$
3. Subtract remainder from $x^r M(x)$. The result, $T(x)$, is the check-summed frame to be transmitted.

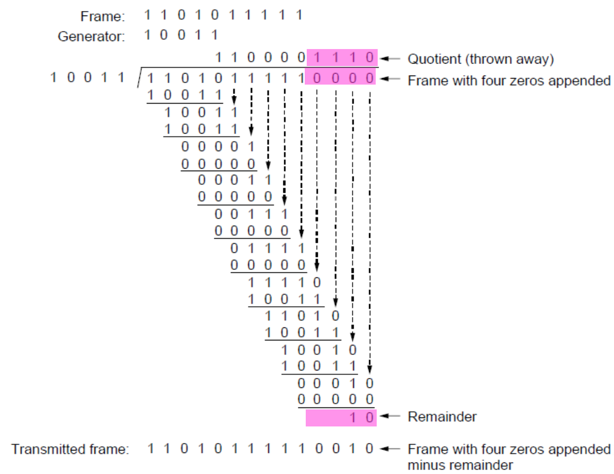
CRC Receiver Algorithm

1. Given T(x) and G(x), compute T(x)/G(x).
   If result is 0 then *no error* detected
   Otherwise, and *error exists*

Example

- Consider a frame M(x)=1101011111 and a generator G(x)=x^4+x+1
- Compute the CRC check-summed frame, *T(x)*

Solution



Khaled Harfoush 2019

Analysis

- In case of errors, T(x) is received as T(x)+E(x)
- The receiver computes (T(x)+E(x))/G(x)
- Since T(x)/G(x)=0, the receiver computes E(x)/G(x)
- The errors will NOT be detected if E(x)/G(x)=0
  - Errors corresponding to polynomials that have G(x) as a factor will not be detected

Khaled Harfoush 2019

Q

- Will CRC detect

1. Single bit errors?
2. Two isolated single-bit errors?
3. An odd number of bits in error?
4. Burst errors of length *r*? Where *r*=number of check bits

1. Single Bit Errors

- $E(x)=x^i$
- If *G(x)* contains *two or more terms*, it will never divide E(x)
- so all single bit errors will be detected

2. Two Isolated Single-Bit Errors?

- $E(x)=x^i+x^j$, where $i>j$
- Can be re-written as $E(x)=x^j(x^{i-j}+1)$
- If we assume that $G(x)$ does *not* divide x (and thus x^j), then errors will be detected if $G(x)$ does *not* divide x^k+1 for all k up to $i-j$
- Polynomials that protect long frames (large k) are known
  - $X^{15}+x^{14}+1$ does not divide x^k+1 for any value of k below 32,768

Khaled Harfoush 2019

3. Odd Number of Bit Errors

- $E(x)=x^i+x^j+x^k$, where $i>j>k$
- Interestingly, *no* polynomial with an odd number of terms has $x+1$ as a factor in the modulo 2 system
- By making $x+1$ as a factor of $G(x)$, we catch all errors with an odd number of inverted bits

Khaled Harfoush 2019

4. Burst Errors of *r* bits

- $E(x)=x^i(x^{k-1}+\ldots+1)$, where i determines how far from right the burst is located

1. If G(x) has an x^0 term, then x^i will *not* have G(x) as a factor
2. If the *degree* of G(x) is *larger* than the *degree* of $(x^{k-1}+\ldots+1)$, then $(x^{k-1}+\ldots+1)$ will *not* have G(x) as a factor

- If conditions 1 and 2 are satisfied bursts of size *r* will be detected

Next Lecture

1. The Data Link Layer -- Protocols