eecs489 COMPUTER NETWORKS

Lecture 27:
Error Control

Error Control

Errors are unavoidable, caused by noise on channel:
- electrical interference, thermal noise, cosmic rays, etc.

Three kinds of transmission errors:
1. sent signal destroyed (doesn't receive data)
2. sent signal changed (received wrong data)
3. spurious signal created (received random data)

Error control: receiver detects and corrects lost or corrupted data
1. error detecting code
2. error correcting code (ECC) or forward error correction (FEC)

Error Control

No error control method is fool-proof

Trade-offs between alternative methods:
- complexity of info computation,
- bandwidth transmission overhead, and
- degree of protection (# of bit errors that can be detected/corrected)

Not often used for mostly reliable links, e.g., fiber, but useful for unreliable links such as wireless
- also used at the transport layer (the Internet is an unreliable "link")

Field: Information Theory

Introduction to Coding Theory

Fundamental issues in information and coding theory:
1. How can we tell when data (transmitted or stored) has been corrupted?
2. How to recover the original data?

Example alternatives:
- do nothing: loss may not be discernable, e.g., concealed by interpolation
- send/store each bit 100 times, majority value accepted as original value
- parity bit: append one single parity bit at the end of message/storage

Introduction to Coding Theory

Main tool and trade-off:

- by sending additional, redundant information, we can detect, and perhaps correct, transmission errors
- the more redundancy, the more effective in error detection/ correction, but the less efficient in bandwidth usage

General idea:

- sender computes some info from data
- sender sends this info along with data

} Transmit extra (redundant) information

- receiver does the same computation and compares it with the sent info

} Use redundant information to detect errors

Examples error detecting code:

- parity check, checksum, cyclic redundancy check (CRC)

Parity Check

- uses an extra bit (parity bit) for error checking
- even parity: total number of 1 bits (incl. the parity bit) is an even number
- odd parity: total number of 1 bits is odd
- single-bit parity examples:

  0100101, even-parity bit =

  0101101, even-parity bit =
- what happens when an error is detected?
  - discard data and, if reliability is required, have sender retransmit
- problem: cannot detect even number of flipped bits

Checksum

Sender treats data as a sequence of 16-bit integers and computes their (1's complement) sum

- transmit the sum along with the packet
- example: 16-bit checksum
  - the string "Hello world." has an ASCII representation of [48 65 6C 6C 6F 20 77 6F 72 6C 64 2E]
  - checksum: 4865 + 6C6C + 6F20 + 776F + 726C + 642E + carry = 71FC

Advantages:

- ease of computation (only requires addition)
- small amount of additional info to carry: one additional 16-bit or 32-bit integer

Used by TCP and UDP

Internet Checksum Example

One's complement arithmetic: when adding numbers, a carryout from the most significant bit is added back to the result

Example: add two 16-bit integers

		1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
		1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
wraparound	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1
1's complement sum		1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
1's complement of sum (Internet Checksum)		0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

Incremental update of checksum [RFC1624]:

`~C' = (~C + m + ~m')`, `m` a 16-bit field of the header

Checksum: Disadvantage

With 16-bit checksum, 1 in 64K corrupted packet will not be detected (probability of a random 16-bit number matching the checksum of a corrupted packet is $1/2^{16}$)

Data Item In Binary	Checksum Value	Data Item In Binary	Checksum Value
00001	1	00011	3
00010	2	00000	0
00011	3	00001	1
00001	1	00011	3
totals	7		7

⇒ under current Internet conditions (error rate etc.), 1 corrupted packet is accepted in in every 300M packets!

Measured on a busy NFS server that has been up 40 days [Mogul92]:

Layer	#errors detected	~#pkts
ethernet	(CRC) 446	1.7×10^8
IP	14	1.7×10^8
UDP	5	1.4×10^8
TCP	350	3×10^7

Cyclic Redundancy Check

Goal of any error detection/correction code: maximize probability of detecting error with minimal redundant info

32-bit CRC protects against most bit errors in messages thousands of bytes long, also used in storage systems (CD, DVD)

CRC is based on finite fields math

Cyclic Redundancy Check

Consider a binary message as a representation of an n-degree polynomial, with the coefficient of each term being 1 or 0 depending on the bit in the message, with the most significant (leftmost) bit representing the highest degree term

- for example: 1011 represents $1x^3 + 0x^2 + 1x^1 + 1x^0 = x^3 + x + 1$

An m-bit message represents a polynomial of m-1 degree

Polynomial Arithmetic

You can divide one such polynomial by another of lower or equal degree by dividing the binary representation of the polynomials, e.g., to divide $x^5+x^3+x^2+x$ by x^3+1, divide 101110 by 1001

Polynomial arithmetic is done using modulo-2 arithmetic, with no carry and borrow: $1+1 = 0+0 = 0$ and $1+0 = 0+1 = 1$, e.g.,

```
10011011       11110000       01010101
11001010 +     10100110 -     10101111 -
---------------  ---------------  ---------------
01010001       01010110       11111010
```

x	y	x XOR y
0	0	0
0	1	1
1	0	1
1	1	0

Note that both addition and subtraction are identical to XOR

Constructing CRC

Let's call the polynomial to be divided T and the divisor/generator polynomial G

Let t be the number of bits in T and $r+1$ be the number of bits in G, $t \geq r+1$

Let's call the remainder of T/G, R; R is of r bits

Want: the polynomial representing the message (M, not T, and not the message itself) to be exactly divisible by $G \Rightarrow$ if the receiver divides the message by G and the remainder is not 0, the message is corrupted

Observe: $M = (T–R)$ is exactly divisible by G, want M

Constructing CRC

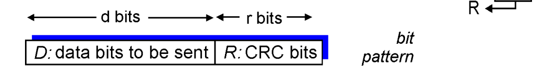
Let D be the message to be sent, e.g., $D = 101110$

Construct T as $D \cdot 2^r$, i.e., D shifted left by r bits,† e.g., $r = 3$, $T = 101110000$

Let $G = 1001$, compute R, the remainder of T/G, by doing long-division with modulo-2 arithmetic, $R = 011$

```
          101011
G <- 1001)101110000 -> D
          1001
           101
           000
           1010
           1001
            110
            000
            1100
            1001
             1010
             1001
              011
         R <- 
```

Construct $M = (T–R) = (D \cdot 2^r – R) = (D \cdot 2^r \text{ XOR } R) = 101110011$; M is exactly divisible by G



†Recall: multiplying a number by 2 is the same as shifting it left by 1 bit

How to Choose G?

Let the string of bit errors introduced be represented by polynomial E

Error will not be detected only if $T+E$ is exactly divisible by G

Want G that makes this unlikely

How to Choose G?

What's known:

- if x^r and x^0 terms have non-zero coefficients, G can detect all single-bit errors
- as long as G has a factor with at least 3 terms, it can detect all double-bit errors
- as long as G contains the factor $(x+1)$, it can detect any odd number of errors
- G can detect any burst (sequence of consecutive) errors of length $< r$ bits

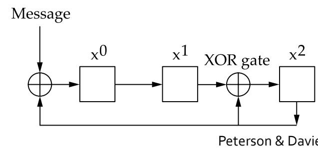
Usually, you just look up a commonly used G

- Ethernet uses CRC-32
- CRC-32's G: $100000100110000010001110110110111$
- CRC-CCITT's G: 10001000000100001

CRC Hardware Implementation

CRC can be cheaply implemented in hardware by implementing the long-division to compute R as a combination of linear feedback shift register (LFSR) and XOR gates, representing G:

- the 0-th term of G occupies the leftmost bit of the shift registers
- each XOR gate represents a modulo-2 addition in G
- the message is fed into the circuit most significant (leftmost) bit first
- each bit of the message causes the current content of the shift registers to be shifted right by one bit
- when the message is exhausted, the shift registers contain R
- for example, computing CRC with $G = x^2+1$ can be implemented as:



Error Correction

Error Correcting Code (ECC) generally requires more redundant bits than error detection

- known in networking as Forward Error Correction (FEC); "forward" because error correction is handled "in advance" before errors occur

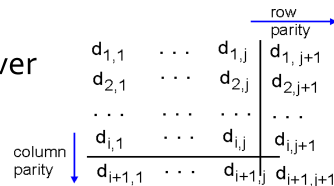
It is usually cheaper to retransmit corrupted data than to transmit redundant data at all times

FEC is most useful when:

1. link is very noisy, e.g., wireless link
2. retransmission will take too long, e.g.,
   - satellite and inter-planetary communication
   - deep space probe transmission
   - real-time audio/video streaming (relatively too long)

2D Parity Check as ECC

- generates both a horizontal or row parity and a vertical or column parity
- both parity info is sent to receiver
- receiver can detect and correct single-bit errors
- problem: cannot detect even number of flipped bits



10101|1
11110|0
01110|1
10101|0

no errors

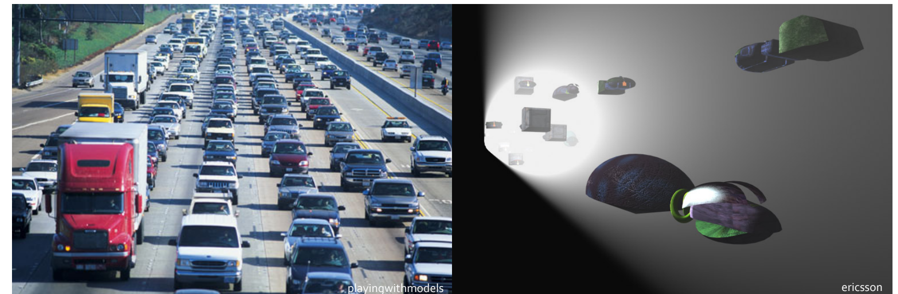
10101|1
10110|0 ← parity error
01110|1
10101|0

parity error

correctable single bit error

Packet Switched Network

Information is transported the same way as cars on freeways: independent data streams may share resources but the information itself is separate



Network Coding

Instead of treating data in discrete, inviolable chunks, network nodes may recombine several packets into one or more output packets

Receipt of information no longer means receiving specific packet content but receiving sufficient number of independent packets

Linear Network Coding

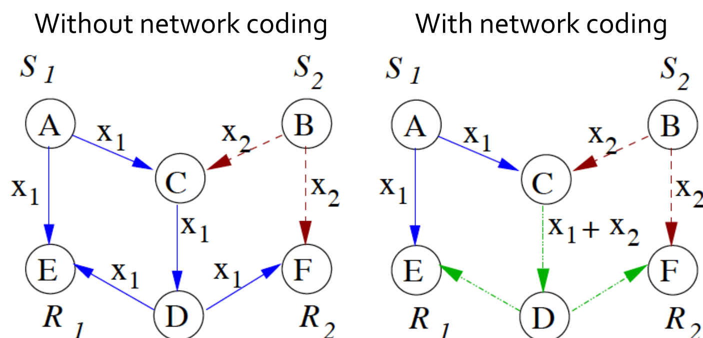
Several packets are linearly recombined into one or more output packets, where addition and multiplication are performed over a finite/Galois field, $GF(2^u)$

- addition is XOR
- multiplication is polynomial multiplication modulo a chosen irreducible polynomial over $GF(2)$
  - can be very efficiently implemented using bitwise operations or 2 log table lookups (see http://www.cs.utsa.edu/~wagner/laws/FFM.html and http://www.ee.unb.ca/cgi-bin/tervo/calc2.pl)

Example: Butterfly Network

Simplest case: S_1 and S_2 want to send to R_1 and R_2

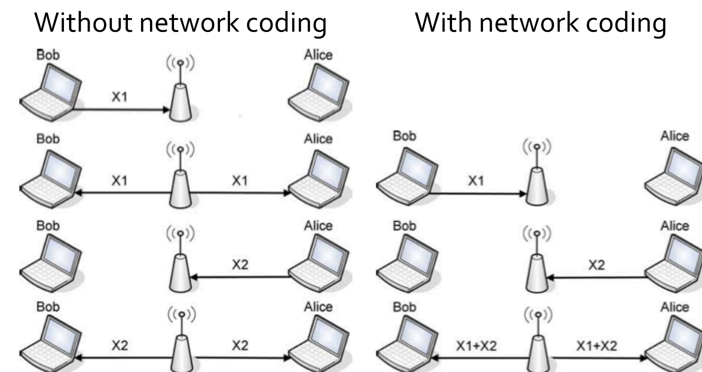
- $u = 1$
- $GF(2) = \{0, 1\}$
- addition is XOR, decoding is also a simple XOR



Use Case: Wireless Network

In the simplest case:

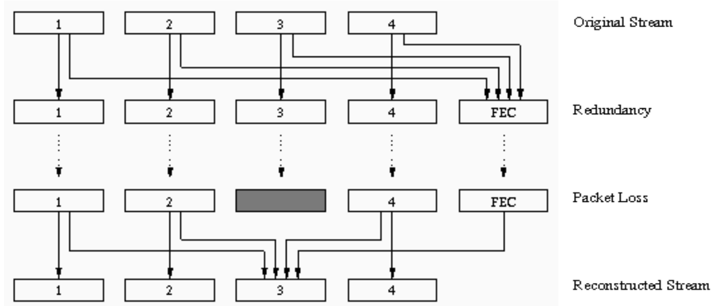
- $u = 1$
- $GF(2) = \{0, 1\}$
- addition is XOR, decoding is a simple XOR



Simple XOR Parity Packet

XOR operation across n packets

- transmit 1 parity packet for every n data packets
- if 1 in n packet is lost, can fully recover



Perkins et al.

Simple XOR Parity Packet

DEMO

Disadvantages:

- delivery to upper-layer must wait for receipt of all $n+1$ packets
- can fix only one lost/corrupted packet

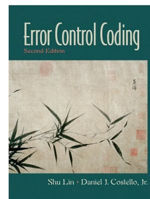
Tradeoff: larger n

- less bandwidth "wastage," but
- longer wait to correct error, and
- higher probability that 2 or more packets can be lost

Reed-Solomon Code

Also based on polynomial codes over finite fields

Good for correcting burst errors



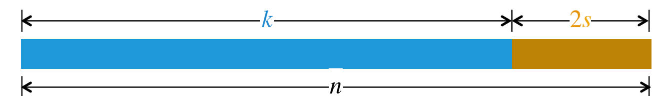
With dedicated hardware, can achieve over 600 Mbps encoding/decoding throughput

Used in CD, DAT, DVD, Blu-ray, Compact Flash, MPEG-2 TS, DSL, RAID, WiMax, DVB, ATSC, the Voyager, Mars Pathfinder, Galileo, Mars Rover, etc.

For more info, take EECS 554 Intro to Digital Communication and Coding or read Lin and Costello

Polynomial-Based Code

A $(n, k, 2s+1)$ code has



k-unit message, where unit is usually in bit or byte

n-unit code word

$2s$-unit parity

can detect $2s$ errors

can correct s errors

Generally can correct α erasures and β errors if $\alpha+2\beta \leq 2s$

Reed-Solomon Code Example

Polynomial representation over $GF(2^8)$:

- message: "hello"
- ASCII decimal: 104 101 108 108 111
- polynomial: $104x^4 + 101x^3 + 108x^2 + 108x + 111$

$\mathrm{RS}(20, 13)$ code over $GF(2^8)$: 13 (k) message bytes, 7 ($2s$) parity bytes: can correct up to 3 errors

- message: "Hello world!\0"
- code word: "Hello world!\08D13F4F94310E5"

[Brown]

Reed-Solomon Code Example

Message:

ALICE'S ADVENTURES IN WONDERLAND
Alice was beginning to get very tired of sitting by her sister on the …

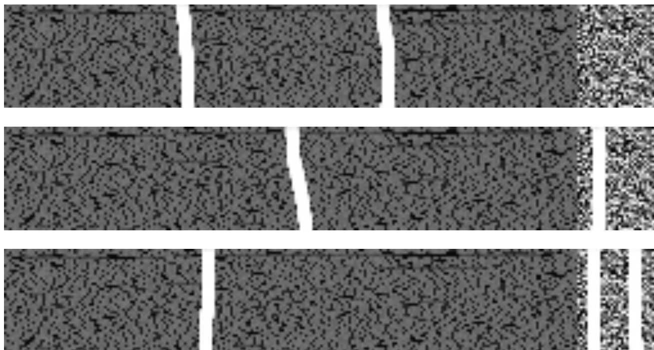
Using $\mathrm{RS}(255, 223)$ code over $GF(2^8)$: visualize each byte as a grayscale pixel such that each row in image is a code word; message is encoded as:

[Brown]

Reed-Solomon Code Example

Since each row is a $\mathrm{RS}(255, 223)$ code word, it can handle up to 16 bytes (or pixels) errors per row

Each of the following still decodes:



[Brown]

Reed-Solomon Implementations

A. Brown's slides:
http://www.cs.duke.edu/courses/spring11/cps296.3/decoding_rs.pdf

R. Morelos-Zaragoza, The ECC Page and source code:
http://www.eccpage.com and http://www.eccpage.com/rs.c

Linux code:
http://www.cs.fsu.edu/~baker/devices/lxr/http/source/linux/lib/reed_solomon/reed_solomon.c

Schifra RS code library:
http://www.schifra.com