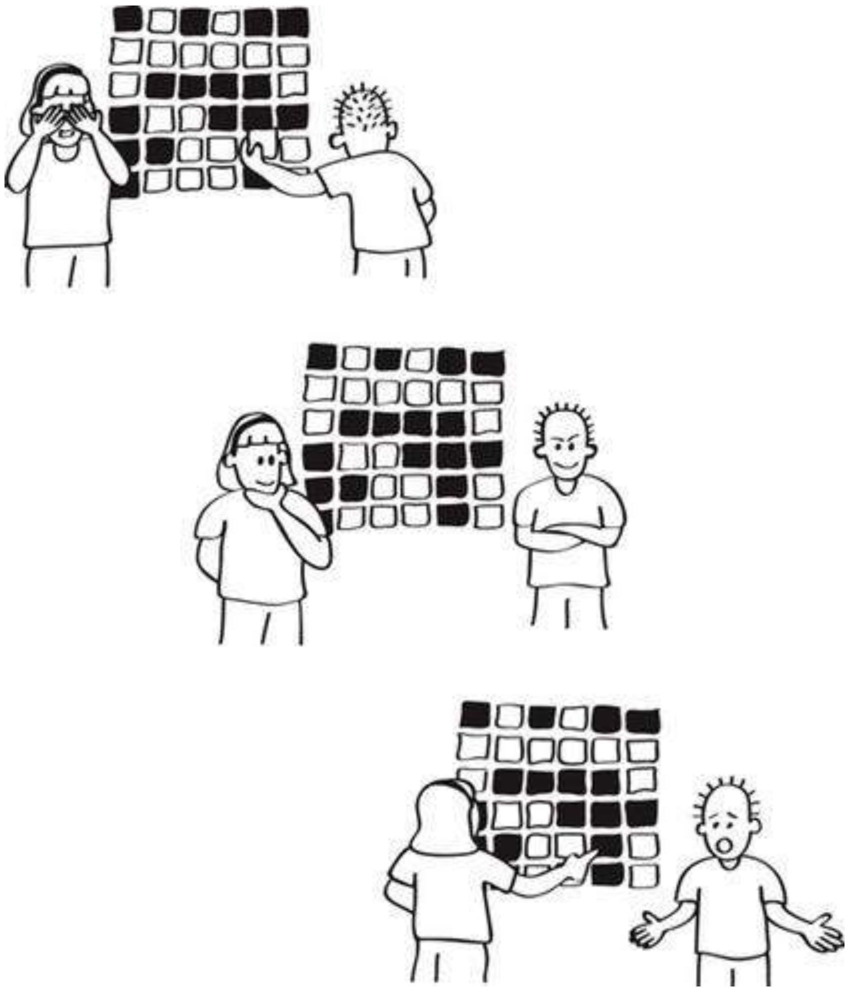
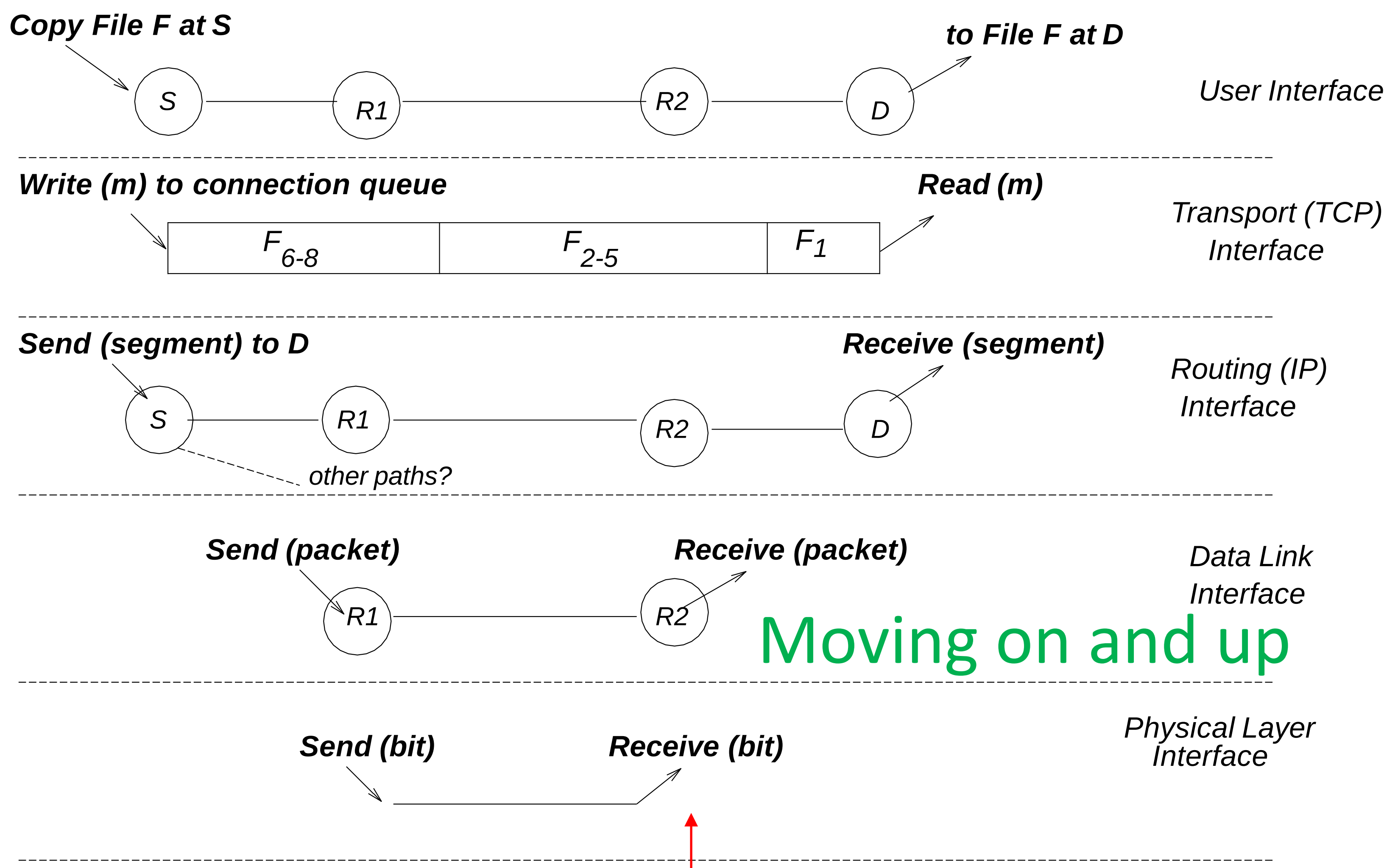
How can we add just 32 bits to a frame and detect almost any error with very high probability

Lecture 6: Error Detection



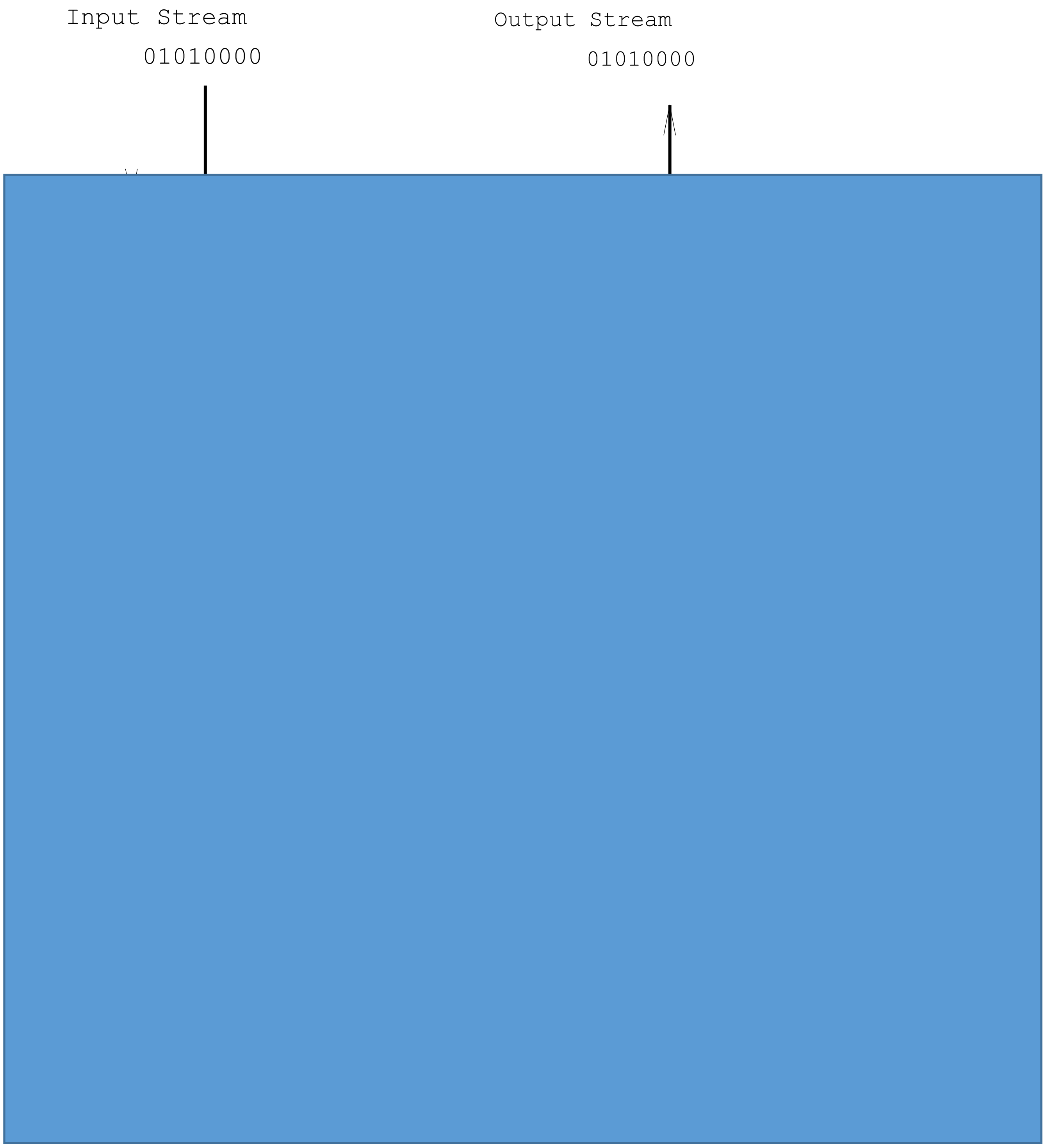


Moving on and up

Been there, done that

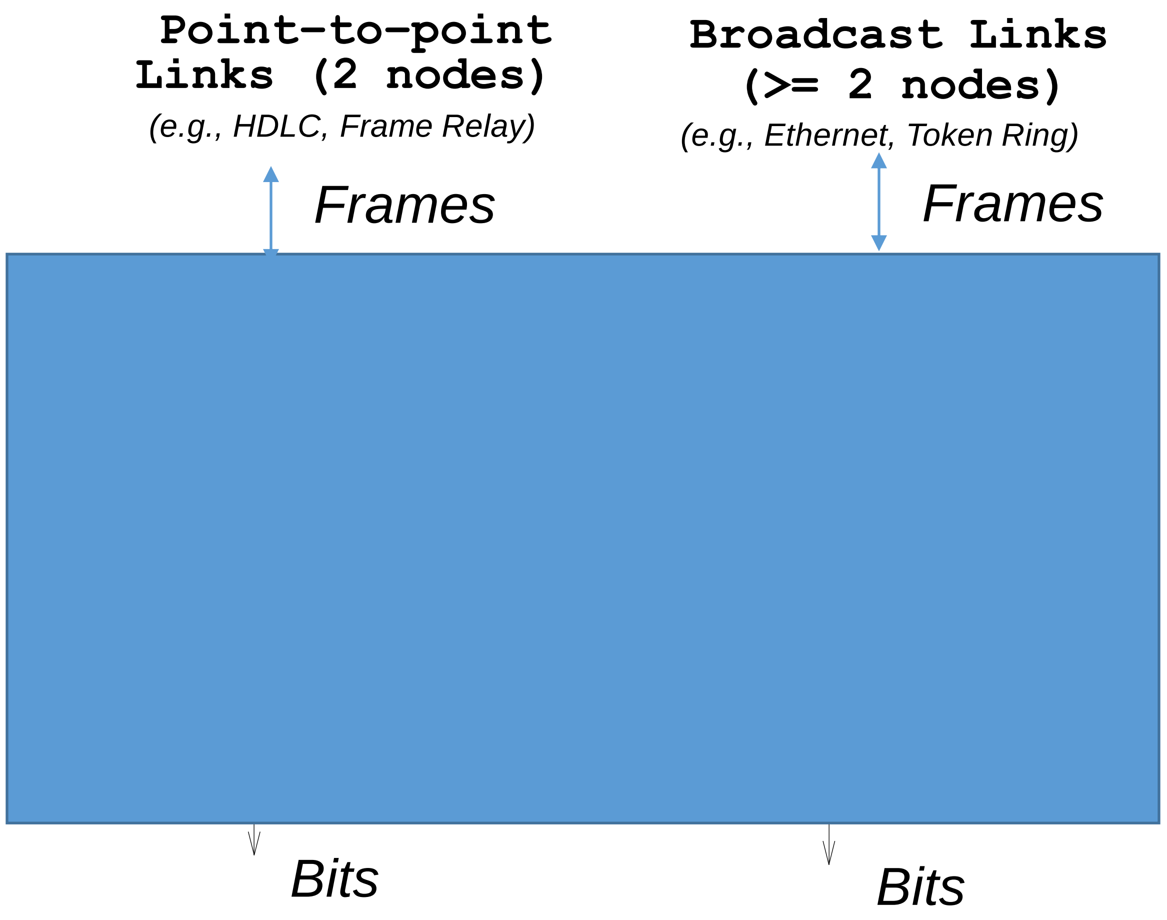
RECALL THE IP ABSTRACTIONS:

PHYSICAL LAYER: SUBLAYERS



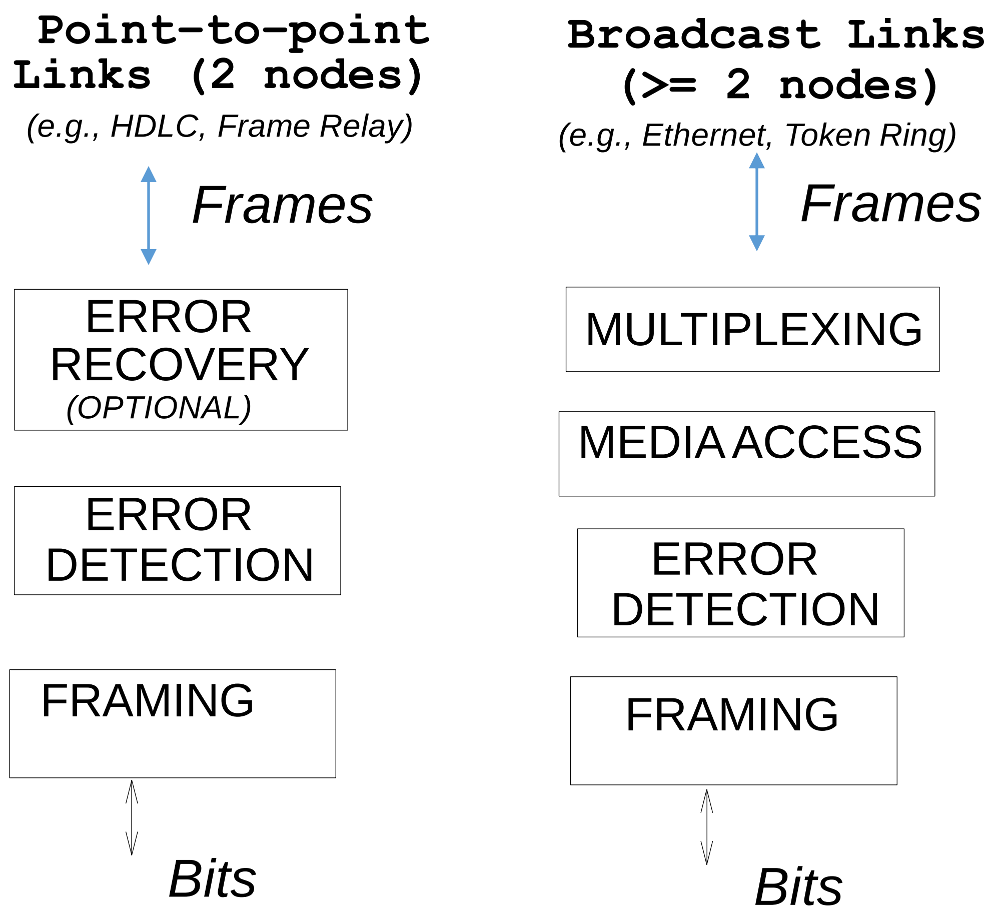
SEMI RELIABLE 1 HOP BIT PIPE

Data Link Sublayers


Point-to-point Links (2 nodes)
(e.g., HDLC, Frame Relay)
Frames
Broadcast Links (>= 2 nodes)
(e.g., Ethernet, Token Ring)
Frames
Bits
Bits

QUASI-RELIABLE 1 HOP FRAME PIPE

Data Link Sublayers


Point-to-point Links (2 nodes)
(e.g., HDLC, Frame Relay)
Frames
ERROR RECOVERY (OPTIONAL)
ERROR DETECTION
FRAMING
Bits
Broadcast Links (>= 2 nodes)
(e.g., Ethernet, Token Ring)
Frames
MULTIPLEXING
MEDIA ACCESS
ERROR DETECTION
FRAMING
Bits

QUASI-RELIABLE 1 HOP FRAME PIPE

Five functions of Data Link

Five functions:

- Framing: breaking up a stream of bits into units called frames so that we can extra information like destination addresses and checksums to frames. (Required.)
- Error detection: using extra redundant bits called checksums to detect whether *any* bit in the frame was received incorrectly. (Required).
- Media Access: multiple senders. Need traffic control to decide who sends next. (Required for broadcast links).
- Multiplexing: Allowing multiple clients to use Data Link. Need some info in frame header to identify client. (Optional)
- Error Recovery: Go beyond error *detection* and take recovery action by retransmitting when frames are lost or corrupted. (Optional)

Goal: Quasi-reliability

- The probability of a receiving Data Link passing up an incorrect frame to the client layer should be very, very small.
  - Undetected error say once in 20 years.
  - Needs to be so small because transport protocols do not insist on end-to-end checksums, a violation of end-to-end argument.
- The probability of a receiving Data Link dropping frames sent by the sender should be small (once a day) to allow good performance.

Types of Errors

- Random Errors. A noise spike or inter-symbol interference makes you think a 0 is a 1 or 1 to 0. Fiber: 1 in 10 ^{10}
- Burst errors: .A group of bits get corrupted because of synchronization or connector plugged in. Correlated!
- Modeling Burst error: Burst error of length k → distance from first to last is k – 1. Intermediate may or may not be corrupted. Burst error of 5 starting at 50. Bits 50 and 54 are corrupted, bits 51-53 **may or may not** be corrupted
- Goal for quasi-reliability: **Like to add checksums to detect as large a burst (say 32) and as many random (at least 3) and any error pattern with very high probability of 1-1/2^{32}**
- Comparison: Imagine a frame of size 1000 and an error rate of 1 in 1000. If random, all frames corrupted on average. If we get a burst of 1000 every 1000 frames, only 1 is lost!

Random versus Burst errors

5

010000010100 Sent

Bit 2 and Bit 6 are corrupted
Can look at as a 2 bit random error
OR a 5 bit burst error

010010100000 Received

5

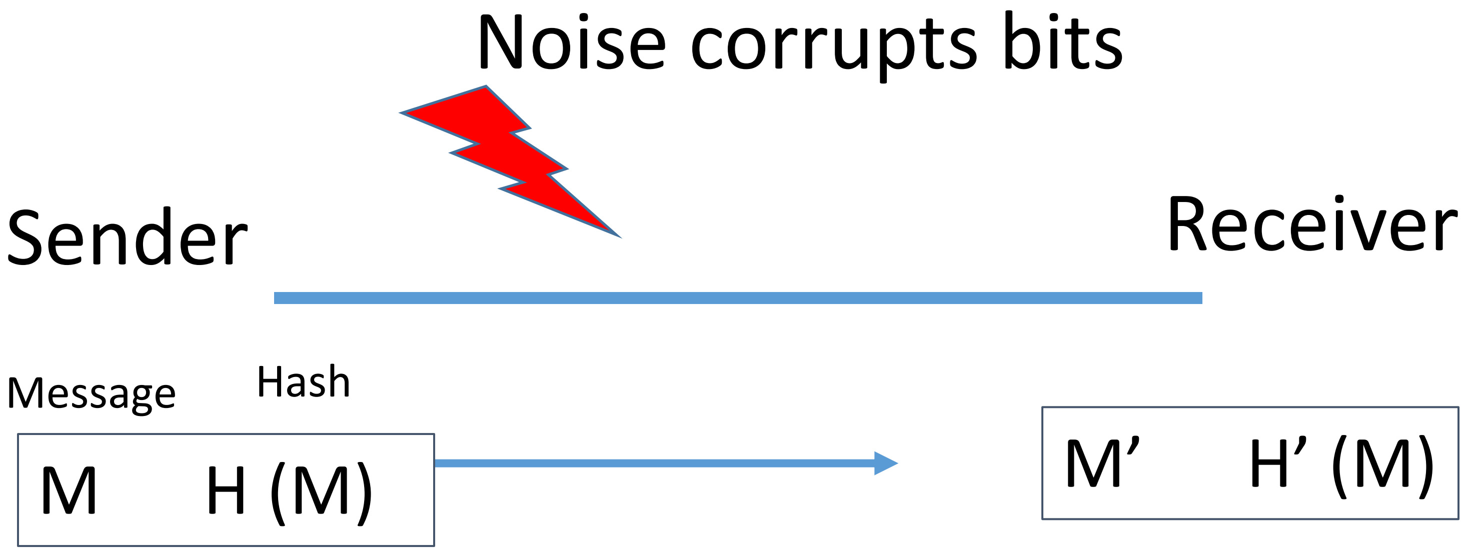
010000010100

3 bit random error but still
a 5 bit burst error

Going beyond Parity

- Parity: ExOR of bits. Can detect all odd bit errors in a frame. Can't detect 2 bit errors. 1011 sent as 10111.
- Would like to do better than parity using so-called checksums for detecting larger number of errors (happens often) by adding more check bits (32 versus 1)

Big picture: checksum as a hash



If receiver finds that H’ = Hash (M’) then receiver
Passes M’ up to higher layer
Else it drops the frame

If M is not equal to M’ and H’ = Hash
(M’) we have an undetected error

But if errors are random and hash is also
random 32 bits then, probability is only $1/2^{32}$

Gentle Intro to Checksums

- Telephone Exchange: Imagine you were sending you were reading a lot of numbers to a friend over a phone. 102, 205, 310 . . . At the end you also send the sum (617)
- Error Detection: **If the sum your friend receives is not the sum of the numbers she received, there is an error**
- Undetected errors: An error that corrupts two numbers so the sum works out. 101, 205, 310, 616
- CRCs: **Instead of being based on addition they are based on division! Also they work Mod 2. Simplest to start with ordinary division and ignore Mod 2.**

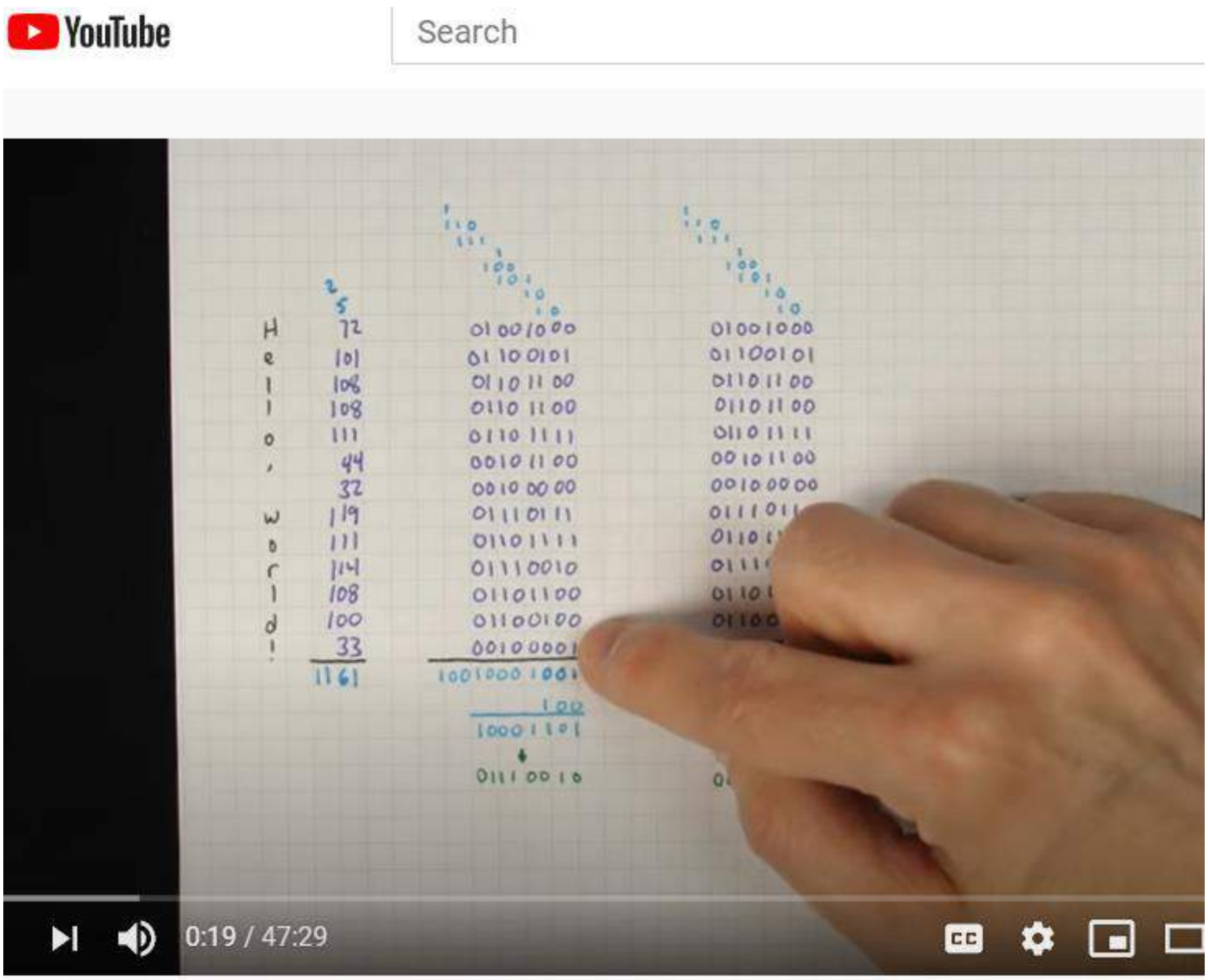
ORDINARY DIVISION CHECKSUM

- Consider message M and generator G to be binary integers.
- Let r be number of bits in G. We find the remainder t of $2^r M$ when divided by G. Why not just M ? So that we can separate checksum from message at receiver by looking at last r bits.
- Thus $2^r M = k.G + t$. Thus: $2^r M + G - t = (k + 1)G$. So we add a checksum $c = G - t$ to the shifted message and the result should divide G.

  Example: M = 110010 (50), r = 3, G = 7. After shifting 3 bits, we get 400. Remainder t = 1, checksum = 6. So we send 110010 110 which is divisible by 7 (406)
- Has reasonable properties. However integer division hard to implement. Prefer to do without carries.

Let’s see Ben explain division checksums

https://www.youtube.com/watch?v=izG7qT0EpBw



H	72	01001000
e	101	01100101
l	108	01101100
l	108	01101100
o	111	01101111
,	44	00101100
	32	00100000
w	119	01110111
o	111	01101111
r	114	01110010
l	108	01101100
d	100	01100100
!	33	00100001
	1161	10010001001

Worth watching up to 6:15

What Ben does well

- Worth checking out the whole video
- Ben uses the same intuition I do in terms of ordinary division. He does the same proofs using polynomials
- His next video on hardware implementation is really cool. I do it on 1-slide
- I think though he misses the fundamental intuition of CRC as a hash function

The Big Idea

- In ordinary division checksums we transmitted a message plus checksum that was divisible by the generator *G*. Thus any errors that cause the resulting number to be not divisible by *G* (invalid codewords) will be detected.
- In CRCs, we do the same thing except that we use Mod 2 arithmetic instead of ordinary arithmetic
- Mod 2 arithmetic: addition and subtraction are XORs so there is no carry. Can implement at high speeds which we need in networks (Terabits)

MODULO 2 ARITHMETIC

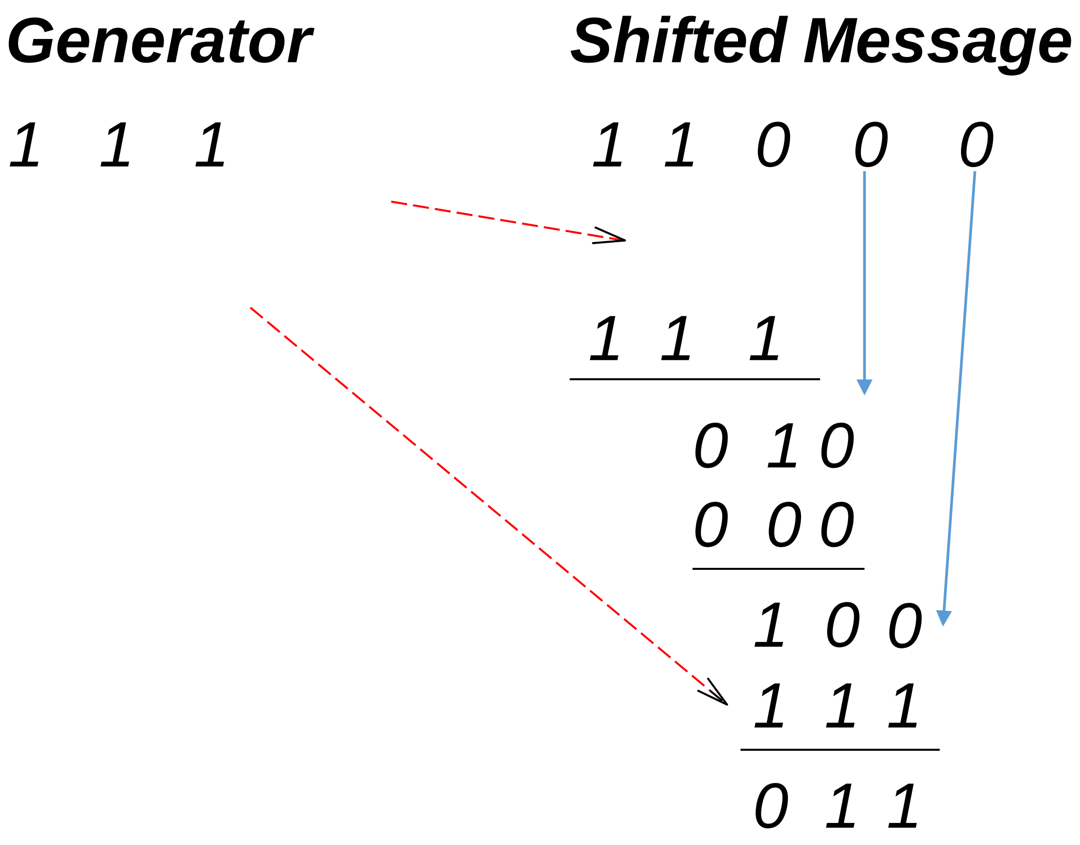
- No carries. Repeated addition does not result in multiplication. e.g. 1100 + 1100 = 0000; 1100 + 1100 + 1100 = 1100
- Multiplication is normal except for no carries: e.g. 1001 $*$11 = 10010 + 1001 = 11011. Shift and Ex-or instead of Shift and Add as in normal arithemtic.
- Similar algorithm to ordinary division. Again let r be number of bits in G. We find the remainder c of $2^{r-1}M$ when divided by G. Why only shift $r-1$ bits this time?
- Thus $2^{r-1}M = k.G + c$. Thus $2^{r-1}M - c = k.G$. Thus $2^{r-1}M + c = k.G$ (addition same as subtraction). Send c as checksum

Recall how ordinary division works

```
          118
        --------------
62      7344
        62
        ----
          114
           62
           ---
            524
            496
           ----
              28
```

- Can be viewed as repeated subtractions of multiples of 62 (i.e., 6200, 620, 496) until we get a number *less* than 62, which is the remainder.

How Mod 2 Division Works

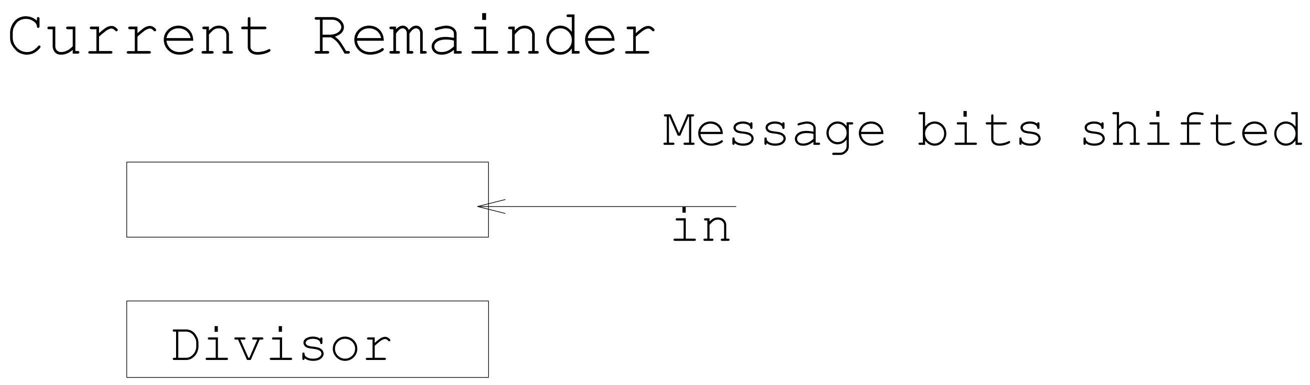


- For CRC, we need to repeatedly add (mod 2) multiples of the generator until we get a number that is $r - 1$ bits long that is the remainder.
- The only way to reduce number of bits in Mod 2 arithmetic is to remove MSB by adding (mod 2) a number with a 1 in the same position.
- While no more bits

  If MSB = 1, XOR with generator (RED)

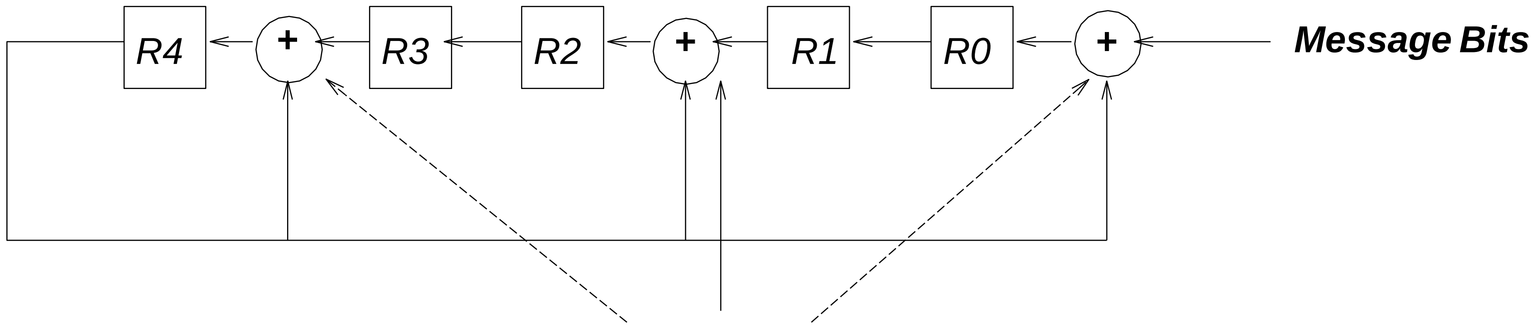
  Shift out MSB and Shift in next bit (BLUE)

Implementing CRCs



- The current remainder is held in a register initialized with first *r* bits of the message.
- If MSB of current remainder is 1, then EXOR current remainder with divisor; if the MSB is 0, do nothing.
- Shift the current remainder 1 bit to the left and shift in next message bit.
- Faster to shift say 8 bits at a time (your assignment) using a table of 256 precomputed checksums of all possible 8 bit checksums

CRC in Hardware: LFSR



Generator String = 1 1 0 1 0 1

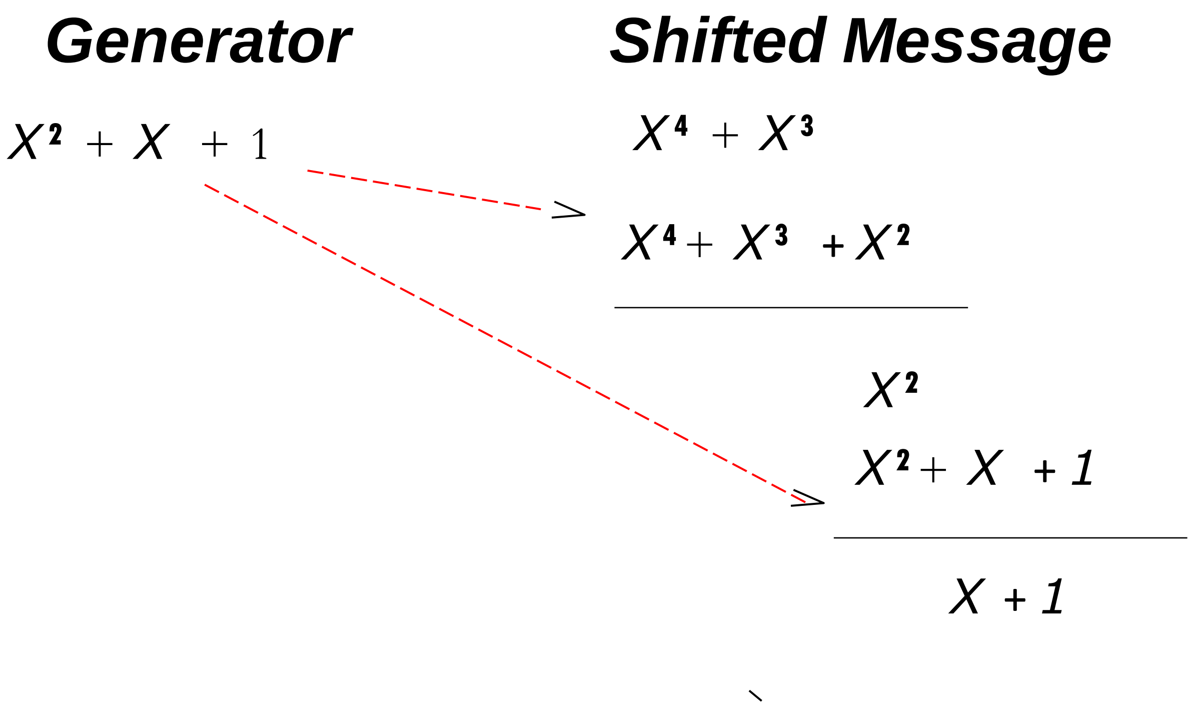
- Registers *R*0 through *R*4 are several single bit registers corresponding to the single multibit register in previous slide.
- Ex-OR placed to right of register *i* if bit *i* set in generator.
- When a message bit shifts in, all registers send (in parallel) their bit values to left, some through Ex-OR gates. Combines left shift of an iteration with MSB check and Ex-OR of next iteration. Ex-OR during left shift.
- Avoids check for MSB: output of *R*4 as input to all Ex-ORs.
- In practice to implement at Terabits we need to shift multiple bits per clock (say 8) cycle using table lookup

So CRCs can be implanted at high speeds. But what errors do they catch? Need a new viewpoint

CRC: Polynomial View

- 101 and 011 can be represented as $X^2 + 1$ and $X + 1$. X^i term iff the i-th bit is 1.
- Normal addition: $X^2 + X + 2$. No carries between powers. $2X$ is bad. Fix by using Mod 2 addition (EX-OR) to get: $X^2 + X$
- Thousand bit message *1001 . . . . 11* becomes: $X^{999} + X^{996} + \ldots X + 1$.
- Can think of CRC computation as dividing a shifted message polynomial (multiplied by x^{r-1}) by CRC divisor polynomial and adding remainder.
- Equivalent to arithmetic (bit) view, but polynomial view is easier to *analyze*.

Division with polynomials



- Same example with bit view (compare with 2 slides back): message is 110, checksum is 111, Shifted message is 11000 which is $X^4 + X^3$
- Same high school algorithm to divide polynomials except subtraction is Mod 2 addition
- Notice we get the same remainder 11.
- Terrible to implement at Terabit speeds but nice to analyze types of errors one can catch

Errors in Polynomial View

Sender **Channel** **Receiver**

100001 → Flip bit 2 and 5 → 000101

- Can model in polynomial view as

Sender **Channel** **Receiver**

$X^5 + 1$ → Add $X^5 + X^2$ → $X^2 + 1$

- In other words, we model bit errors in positions i, j, and k etc. as the channel adding an error polynomial $X^i + X^j + X^k + \ldots$
- Since the message polynomial divides the generator the error will not be detected iff the error polynomial does not divide the generator.
- Or in other words if one cannot multiply the generator by another polynomial to get the error polynomial. Undetected errors are bad news!

CRC PROPERTIES

CRC-16: $X^{16} + X^{15} + X^{2} + 1 = 11000000000000101$

Error results in adding in a polynomial. Use normal polynomial *multiplication* intuition.

Single bit errors: result in addition of x^{i} say x^{1000} If $G(x)$ has at least two terms, any multiple of $G(x)$ will have two terms. Multiplying CRC-16 by $X^{100} + X^{10}$ will have at least two terms: $X^{116} + X^{10}$

Two bit errors correspond to adding $x^{i} + x^{j}$, which will not divide if $G(x)$ does not divide $x^{k} + 1$ for sufficiently large k. Done by design. See notes

Odd bit error polynomials are never divisible by $x + 1$. Why? If P(X) = (X + 1) R(X) then if we plug in 1, we get P(1) = 1 on LHS but 0 on RHS. So make G have $x + 1$ as a factor to detect odd bit errors.

But what makes CRCs shine are detecting burst errors as we will see next.

CRC AND BURSTS: THE BIG DEAL

Consider a 1000 bit message with a 16 bit burst error starting in bit 900

The error polynomial could be something like:

$X^{900} + X^{890} + \ .\ .\ X^{885}$

Factoring we get:

$X^{885}\ (X^{15} + X^{5} + \ .\ .1)$

But $X^{15} + X^{5} + \ .\ .1$ cannot divide CRC-16 which is : $X^{16} + X^{15} + X^{2} + 1$. Thus CRC-16 can catch any 16 bit or smaller burst

In general, burst errors of length k adds $x^{i}(x^{k-1} + ..1)$. Can catch if $k \leq$ generator degree. Any multiple of generator will have a term of x^{k} or higher

So CRC-32 can catch any 32 bit burst error for sure but we will see in HW that it will only miss larger burst errors with very small probability: $1/\ 2^{32}$

Back to the big picture

- Remember the main question. How can we add just 32 bits and catch all errors with low probability
- Answer: CRC is a great hash function (remember the minflip) so chances of a random error fooling this hash is $1/2^{32}$.
- But there's more: the polynomial structure allows CRC-x to catch all .x bit burst with probability 1, and up to 3 random errors
- Implementation Best of all, it can be implemented in hardware using LFSRs at Gigabit speeds., When do we an end-to-end checksum at TCP in software we do a more wimpy sum based checksum.

Hamming Distance and CRCs

- Recall we used CRCs to do better than parity using so-called checksums for detecting larger number of errors (happens often). An idea called Hamming Distance explains why some codes detect and correct more bits.
- Hamming Distance between two strings S and R is the number of bit positions they differ. Thus Hamming Distance of 11011 and 10111 is 2.

Error Detection as Coding

- Computation View: add bits to the end of a frame for detecting errors. Parity adds 1, CRC adds 32
- . Coding View: We take every packet P and encode it as a frame F = C(P) in any way you wish. One way is to add a checksum but there are other examples.
- Triple Code: Imagine we encode every bit in P as 3 identical bits in F. 10 would encode as 111000.
- Coding View after coding some codewords are **valid** and some are **invalid**. For example , in the Triple Code if we receive 101000 we know its an invalid codeword
- So what? The coding view allows us to abstract across details of code and use Hamming distance.

CRCs and Hamming Distance

- While CRCs are really chosen for burst detection, their Hamming distances are also important
- CRC-32 layers has a Hamming Distance of 5 for frame sizes less than 360 bytes and 4 for larger frames. CRC-32 is used in 802.11, Ethernet, SATA etc.
- This implies that CRC-32 can detect 4 random bit errors for most frames
- There is a special 32 bit polynomial called CRC32K by Koopman that has Hamming distance of 6 for fairly large frame sizes

Random Error Detection Capability of CRC-32

https://users.ece.cmu.edu/~koopman/crc/

Note: This is enrichment only, not for Midterm!

R.W. Hamming

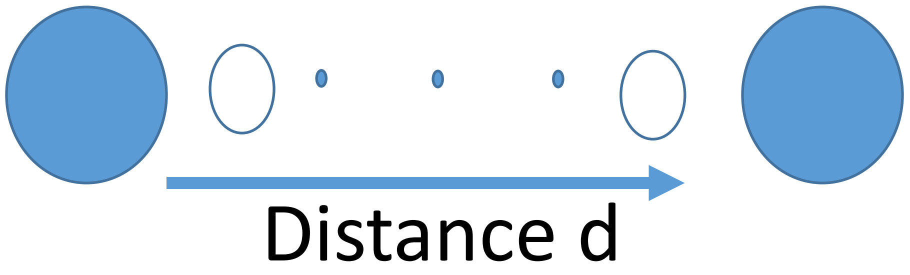
At Bell Labs at first I sat with the mathematicians . . . then the physicists . . then the chemists

My own lesson: collect different points of view. Example: coordinate geometry turns geometry into algebra

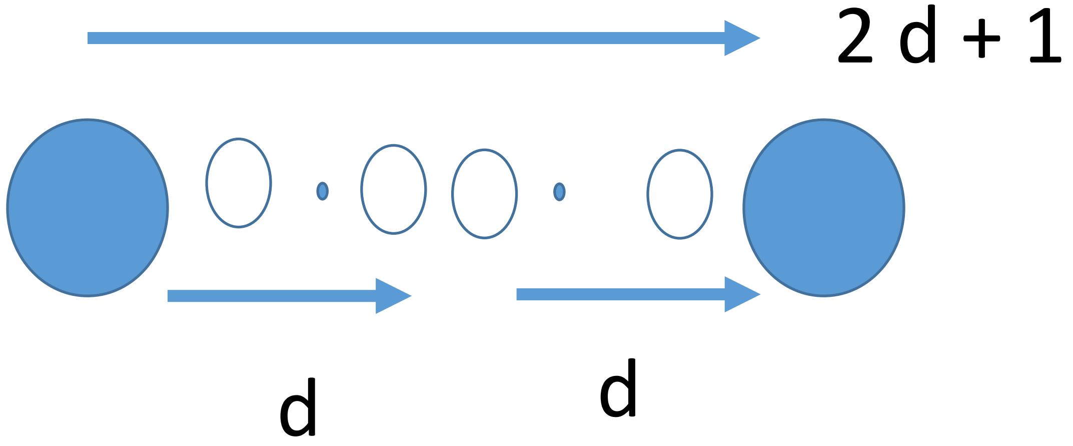
Coding to Detection/Correction

- Hamming Distance and Codes: Hamming distance is the smallest number of errors to convert a valid code word C to another valid code C' causing a mistake.
- Prediction: Use hamming distance to see how well a scheme can detect and even correct errors
- Detection: code tells you some bit in frame has been corrupted, Response: drop frame and retransmit later
- Correction: The code .tells you which bit is in error so you can correct it. Response: flip that bit
- Question: Computer disks corrupt bits but use error correction while networks use detection. Why?

Hamming Distance, Detection, and Correction



- Can *detect d* random bit errors if Hamming distance is. *d* + 1 because flipping *d* bits cannot move from valid codeword (blue) to another.
- Can *correct d* errors if Hamming distance is 2*d* + 1. Can draw a “ball” of radius *d* around each valid codeword *C* and assign invalid codewords within ball to *C*. Can also *detect* 2*d* errors with same code but cannot do both correction and detection at same time!



Hamming Distance in action

- Triple Code: Hamming distance is 3 and can either correct 1 (e.g., 110 -> 111) or can detect 2 errors but not both
- Parity: Hamming Distance is 2 and can only detect 1 bit errors
- Networks use only error detection: error correction too expensive. Parity cheap but inadequate,
- CRCs: We now describe the form of error detection all networks use called CRC that can detect large burst errors (Its claim to fame) and almost all random errors.
- Note: Hamming distance gives you no insight into burst error detection which is the real big deal for CRCs

Lessons from Framing and CRCs

- End-to-end argument.
- Sublayering is a powerful tool: bit stuffing implementation, error recovery on top of framing. Sublayers extract their penalty
- Common problems at layers and exploiting solutions at other layers: coding, bit and frame synchronization, getting extra symbols from physical layer.
- Arguing by Analogy: ordinary division and CRC. Helps when trying to do CRC multiple bits at a time.

General and abstract approaches help: error detection in terms of coding.

- Having Multiple Views: Bit string view for CRC computation and polynomial view for analysis.