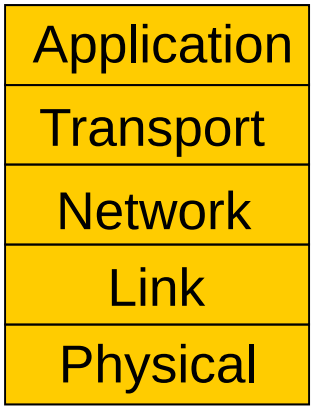
CSEP 561 – Error detection & correction

David Wetherall
djw@cs.washington.edu

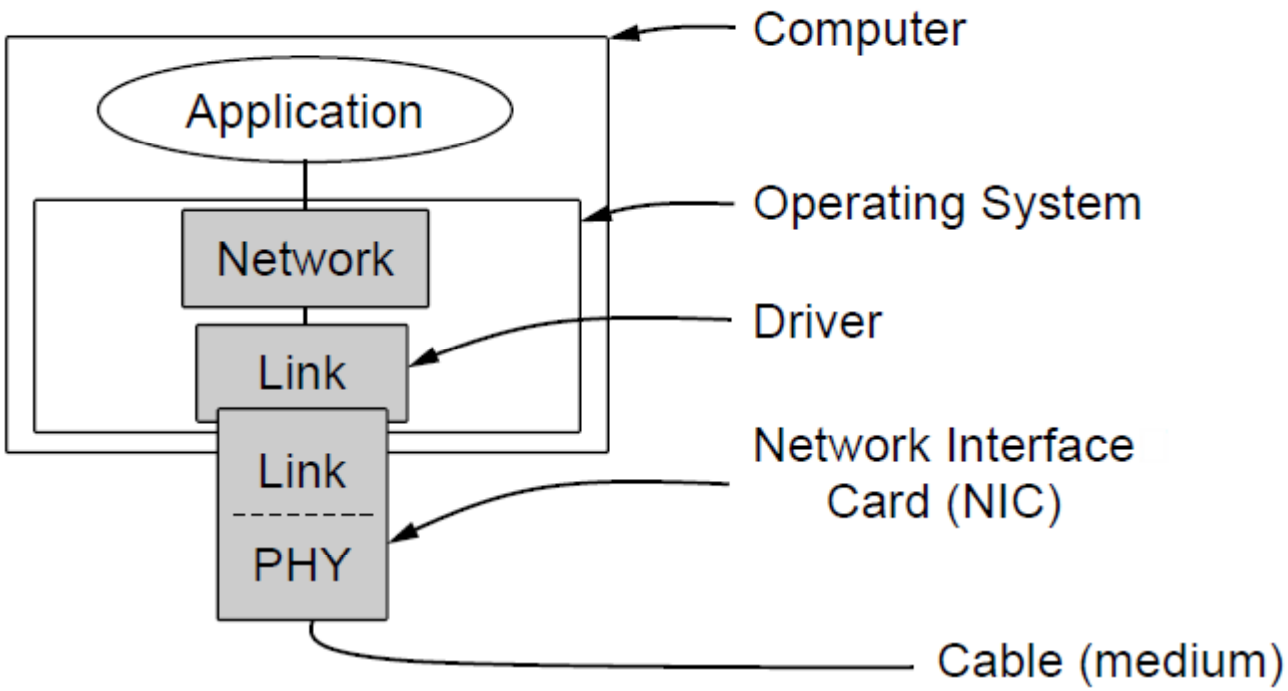
Codes for Error Detection/Correction

- Error detection and correction
  - How do we detect and correct messages that are garbled during transmission?

- The responsibility for doing this cuts across the different layers
  - But we're mostly thinking about links

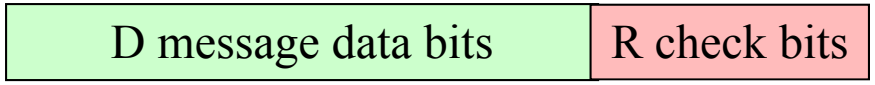


Typical implementation of the layers



Codes for Error Detection/Correction

- A scheme maps D bits of data into D+R bits – i.e., it uses only 2^D distinct bit strings of the 2^{D+R} possible.



(systematic)

- The sender computes the check bits based on the data.
- The receiver also computes check bits for the data it receives and compares them with the check bits it received. Mismatches detect errors. And mapping to the "closest" valid codeword can correct errors.
- Detection/correction schemes are characterized in two ways:
  - Overhead: ratio of total bits sent to data bits, minus 1
    - Example: 1000 data bits + 100 code bits = 10% overhead
  - The errors they detect/correct
    - E.g., all single-bit errors, all bursts of fewer than 3 bits, etc.

The Hamming Distance

- Errors must not turn one valid codeword into another valid codeword, or we cannot detect/correct them.

- Hamming distance of a code is the smallest number of bit differences that turn any one codeword into another
  - e.g, code 000 for 0, 111 for 1, 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., 001 → 000

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.

- Q: What kind of errors will/won't checksums detect?

Internet checksum properties

- Catches all error bursts up to 15 bits, most 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

Fletcher – a better checksum

- Includes a "positional component"

$$\begin{aligned}&\text{Initial values :} && sumA = sumB = 0;\\ &\text{For increasing } i : && \{\ sumA = sumA + D_i;\\ & && \ \ sumB = sumB + sumA;\ \}.\end{aligned}$$

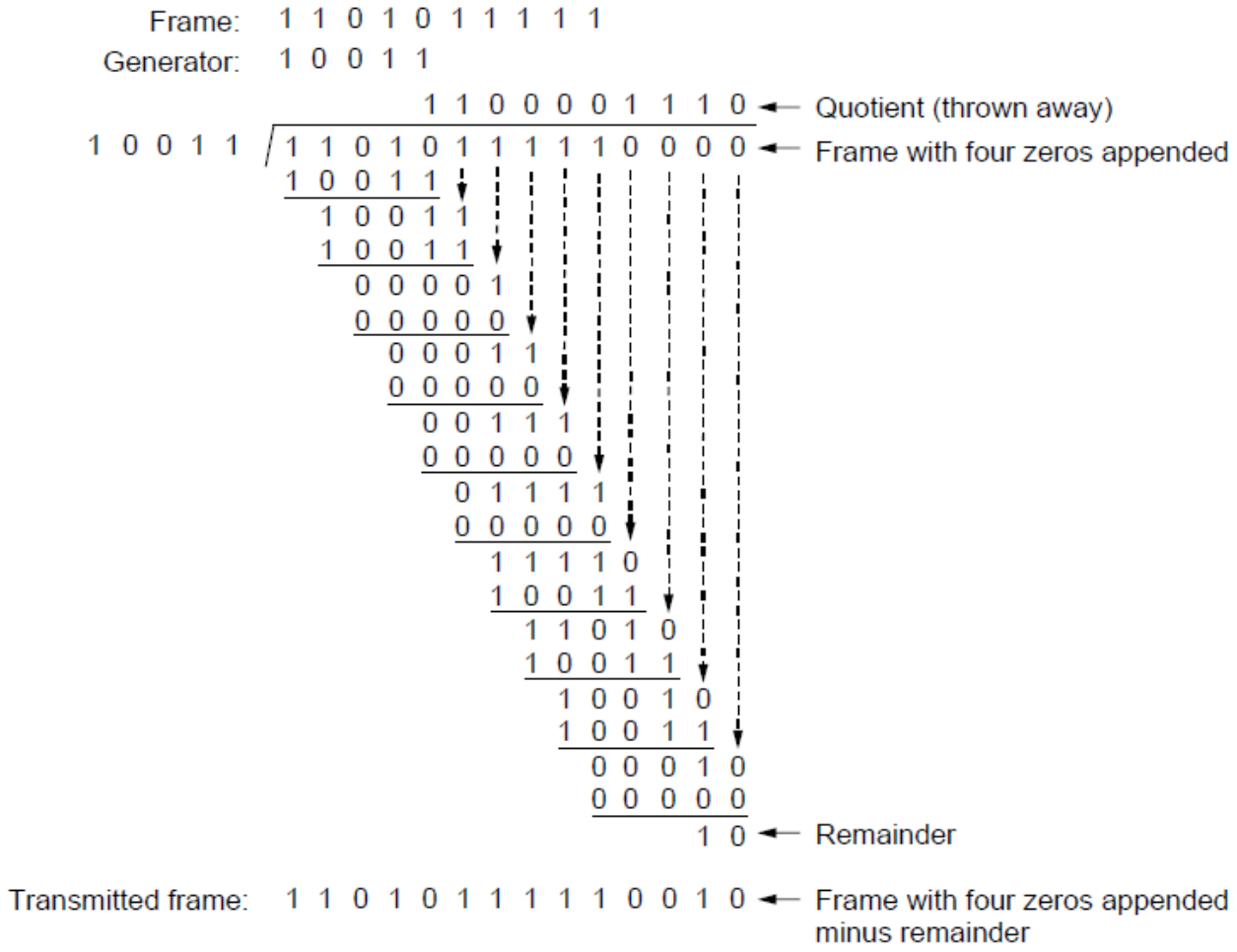
- Now sensitive to order of data
  - slightly more computation, but well worth it

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 computation example (from book)

- Yuck!
- Do this in hardware with XOR & shifts



“Standard” CRC-32

- It is
  - CRC-32: 10000010011000001000111010110111
  - 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

A better CRC

- Castagnoli, Koopman
  - Via exhaustiv search!

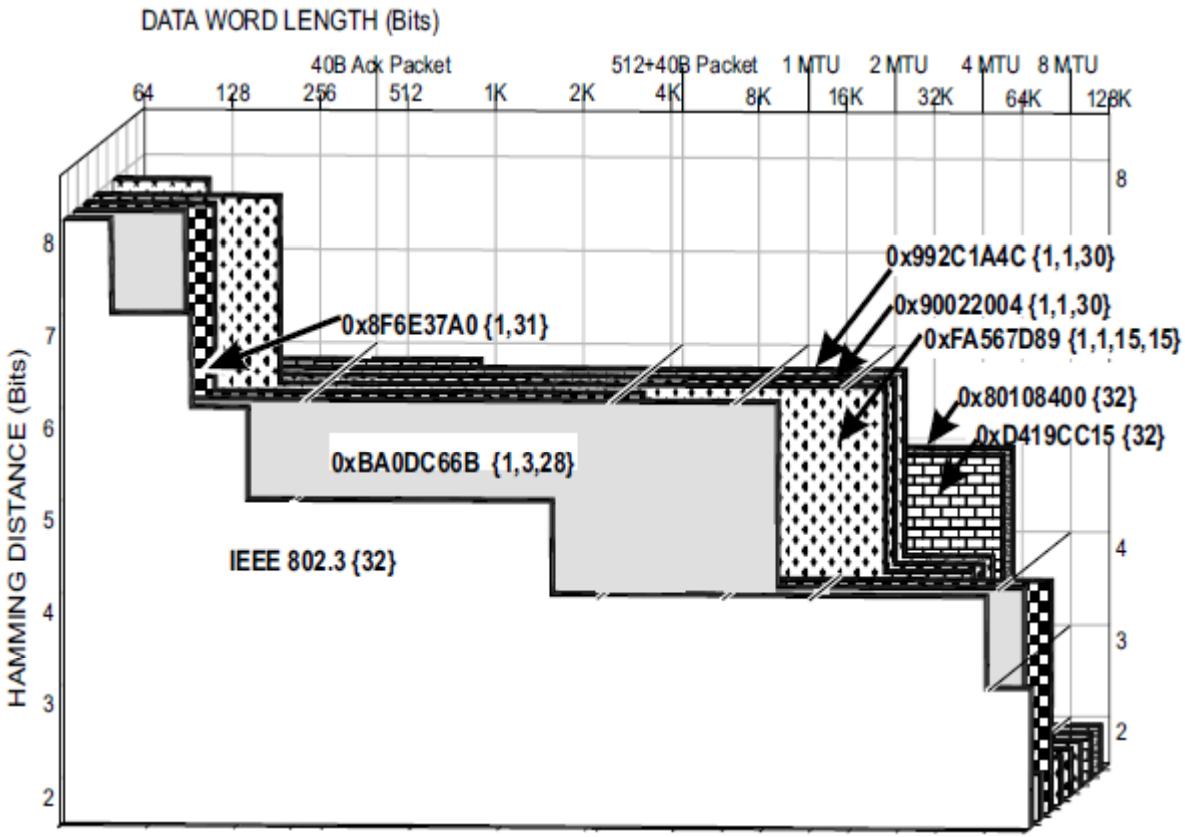


Figure 1. Error detection capabilities of selected 32-bit CRC polynomials.

Real Error Detection/Correction codes

- Detection (often at link/network/transport layers)
  - Parity, simple example
  - Checksums, but weak
  - CRCs, widely used

- Correction (often at physical and application layers)
  - Hamming codes, simple example
  - Convolutional codes
  - Reed-Solomon / BCH
  - Low-density Parity Check (LDPC) codes ← future

- Based on mathematical properties …

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 $\leq 2\%$

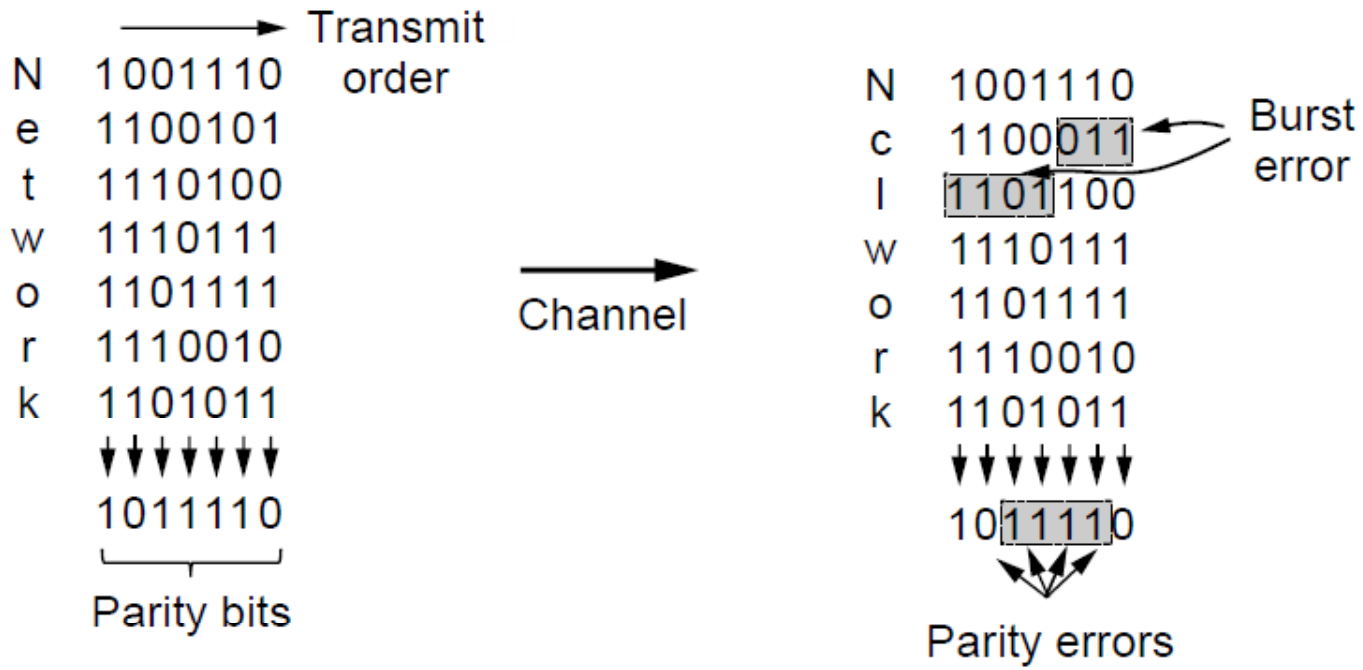
Real Error Models

- 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

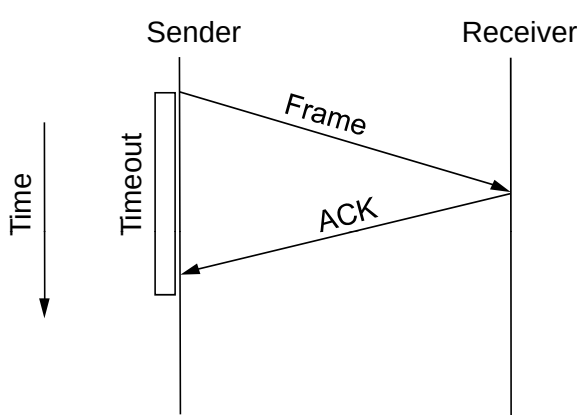
- For bursty errors, either want:
  - A code that is built to handle them well
  - To convert them to random errors (interleaving)

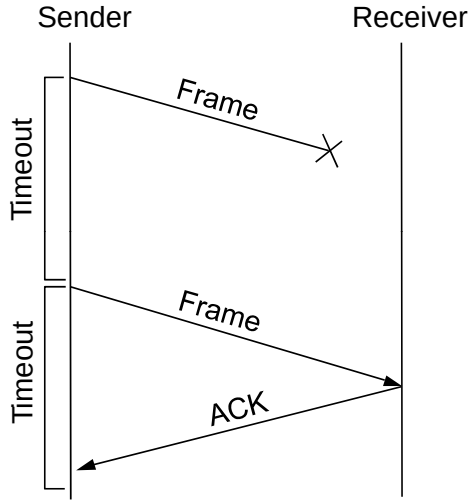
Interleaving – a neat trick

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



Retransmissions, or more formally Automatic Repeat Request (ARQ)



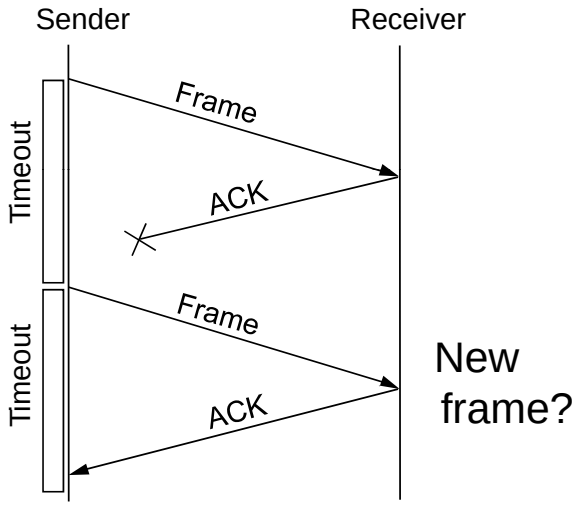


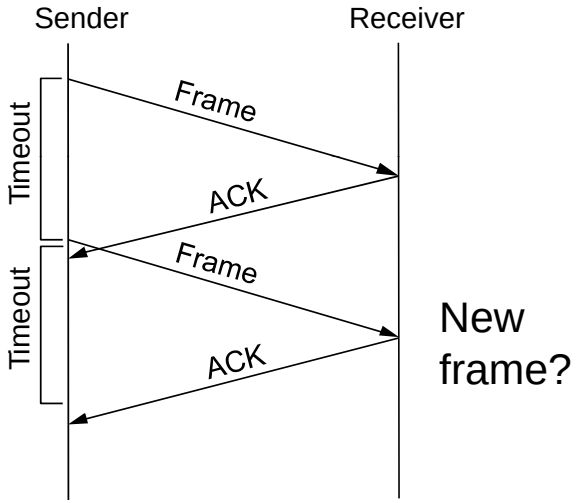
- Packets can be corrupted or lost. How do we add reliability?
- Acknowledgments (ACKs) and retransmissions after a timeout
  - Automatically resend until a positive acknowledgement is received
- ARQ is generic name for protocols based on this strategy

Two issues

1. How long to set the timeout?
   - Only easy on a direct link, otherwise timing variability
   - Way too long lowers performance
   - Implies sometimes timeout will be early
2. How to avoid accepting duplicate frames as new
   - Given retransmissions, frame loss, and imprecise timeouts

The Need for Sequence Numbers

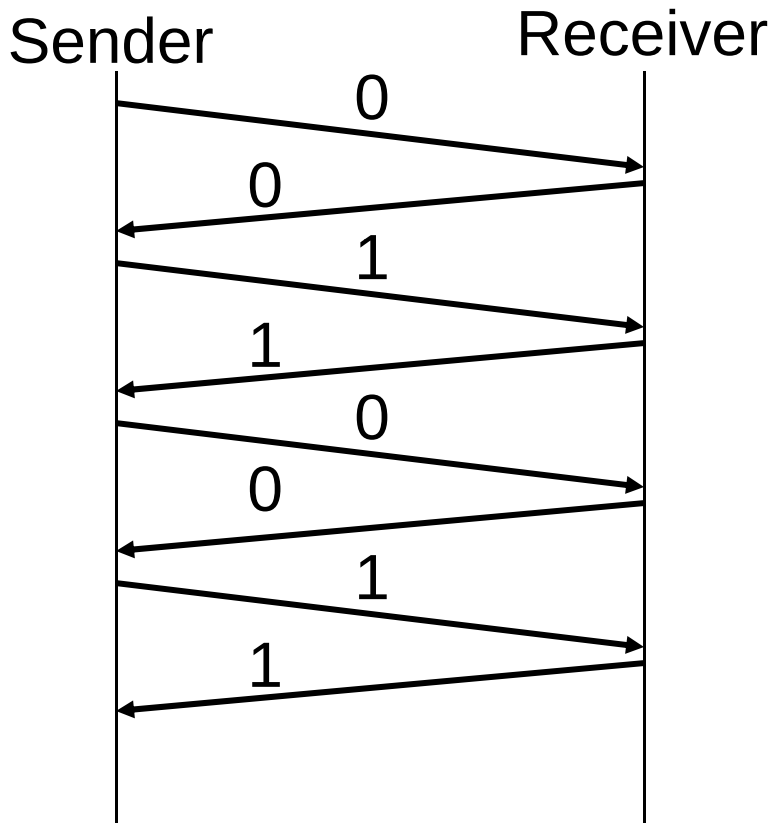




- In the case of ACK loss (or poor choice of timeout) the receiver can't distinguish this message from the next

Stop-and-Wait

- Only one outstanding frame at a time, 0 or 1.
- Retransmissions re-sent with same number
- Number only needs to distinguish between current and next frame
  - A single bit will do



Limitation of Stop-and-Wait

- Lousy performance if wire time << prop. delay
  - How bad? You do the math
- Want to utilize all available bandwidth
  - Need to keep more data “in flight”
  - How much? Remember the bandwidth-delay product?
- Leads to Sliding Window Protocol (later)

Error Correction

- Two strategies to correct errors:
  - Detect and retransmit, or Automatic Repeat reQuest. (ARQ)
  - Error correcting codes, or Forward Error Correction (FEC)

- Question: Which should we choose?

Example: 802.11

- The standard scheme is:

- Link: 32 bit CRC on frame and retransmission
- PHY header has 16 bit CRC
- PHY: FEC on data via interleaving and a binary convolutional code or LDPC
  - rates from ½ to 5/6.

ARQ vs. FEC

- Will depend on the kind of errors and cost of recovery
- Example: Message with 1000 bits, Prob(bit error) 0.001
  - Case 1: random errors
  - Case 2: bursts of 1000 errors

- FEC used at low-level to lower residual error rate
- ARQ often used at packet level to fix large errors, e.g., collision, loss, as well as protect against residual errors
- FEC sometimes used at high level, e.g.:
  - Real time applications (no time to retransmit!)
  - Nice interaction with broadcast (different receiver errors!)

3. Maranello and costs/benefits

- Throughput benefit:
  - Often this is the precious resource; the primary benefit of the design is increased throughput (+30% on average)
- Latency benefit:
  - Recovery from error is faster

- Computation cost:
  - More processing / complexity for all good packets, not just bad
  - Consumes more energy, e.g., handhelds, not always desirable

- Message cost:
  - Can be longer (NACK plus retransmit), but not likely

2. Maranello and the BER

- The bandwidth advantage of Maranello comes when "most" blocks in a frame are not errored
  - These blocks cost only a 4-byte checksum, not a 64-byte resend
- So the pattern of errors matter, not just the average BER.
  - 1% BER with random errors → every 64 byte block has errors
  - 1% BER in bursts of 10 → most blocks are not in error
- Bursty errors are more likely in practice

1. Maranello and deployment

- Some different kinds of compatibility:
  - M(aranello) devices on different networks than V(anilla) ones
  - Can mix M and V devices on one network, but they can't talk
  - Can freely mix M and V devices on one network
  - Maranello supports the latter; highest level of deployability

- Key assumption:
  - M devices send valid V messages, including timing
  - But this requires V devices to ignore "new" messages and M devices to work even if a new message is ignored.
  - More generally, may negotiate capabilities