Error detection: Outline

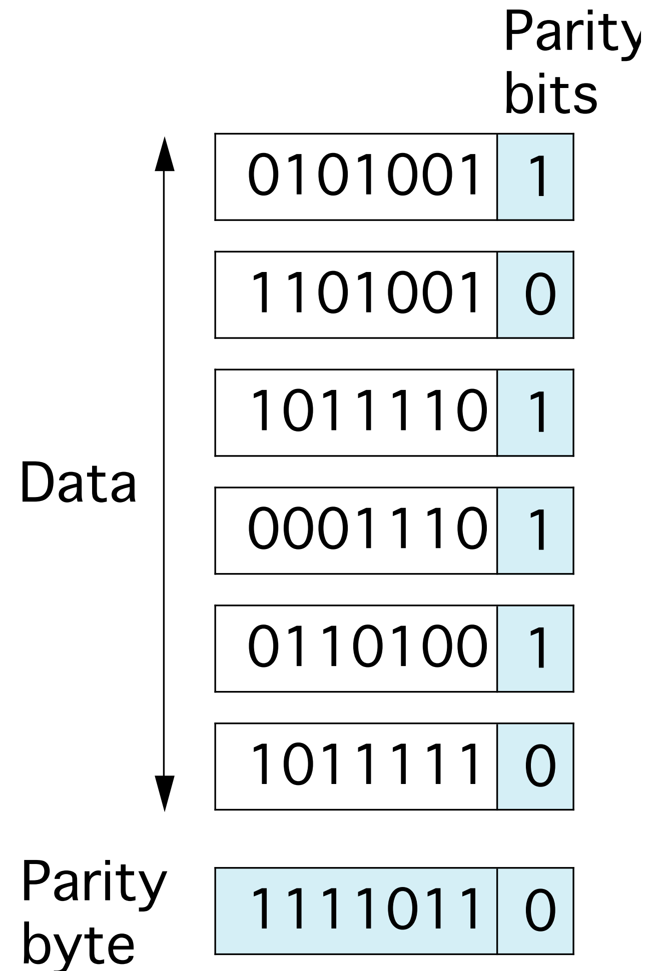
- In the last class, we looked at the problem of encoding bits on the wire and framing to delineate when a frame begins and ends.
- Today we look at how we detect errors introduced by the network
  - Mechanisms depend on how much computational overhead is tolerable, how much extra bits are wasted and what types of errors (number of errors, error types etc.) have to be detected
- First problem is to detect errors. The second problem is to correct errors
  - Correction can be achieved by resending the frame
  - Or by sending extra bits so that the receiver can reconstruct the erroneous frame

Error detection and correction.

- Parity or checksums
  - Send additional bits that can help identify if there was an error in the transmission
  - The goal is to keep this extra bits as small as possible
- One dimensional parity
  - Add one extra bit to a 7-bit code to keep either a odd- or even- number of 1s in a byte
- Two dimensional parity
  - Calculates parity across all bit (in a given bit position) in the frame
  - This uses an extra parity byte

Two dimensional parity

- It can be shown that two dimensional parity catches all 1, 2 and 3 bit errors and most 4-bit errors
- In this example, we added 14 bits of redundant information to a 42 bit message



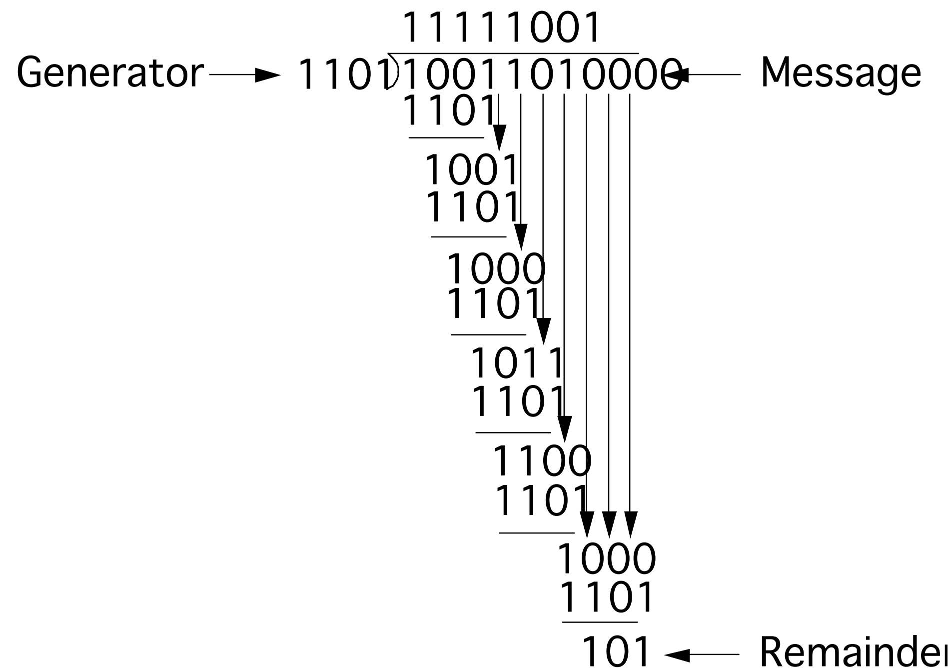
Internet checksum

- Add all the bytes and then transmit the sum.
  - Receiver does the same summation and checks the sum. If they don't match, then there was an error
- Internet checksum:
  - Consider data as 16-bit integers. Add them using 16-bit ones complement arithmetic
    - In ones complement, negative number is represented each bit inverted
    - Ones complement addition, carryout from most significant bit is added to result
  - Take ones complement of the result
  - Resulting 16 bit number is the checksum
- Overhead is 16 bits per message
- Internet checksum is simple but does not detect many errors - used in conjunction with others

Cyclic Redundancy Check (CRC)

- Fairly intensive computation
  - 32 bit CRC can check errors for a longer message
  - Tradeoff between computational complexity and check requirements
  - CRCs are based on finite fields
- Assume (n+1) bit message as a polynomial of degree n. Choose a CRC polynomial C(x)
  - When transmitting message M(x) of size, transmit k extra bits such that the new message P(x) is exactly divisible by C(x)
    - want k << n
    - e.g., in Ethernet, k = 32 and n = 12,000 (1500 bytes)
  - Transmitted message is P(x)
  - Receiver does the same, divide P(x) with C(x). If there is no remainder, then there was no errors

- Polynomial arithmetic modulo 2
  - If polynomials of same degree, then they divide
  - Subtraction is basically a xor operation
    - Xor is 1 if the two bits are different (0 & 1 or 1 & 0)
    - Consider M(x)=1001101 and C(x)=1101 and k=3



- Key is to choose C(x) such that common errors are caught
  - CRC-8: 100000111
- Each CRC function has different strengths in detecting error conditions
  - E.g. all single-bit errors, as long as xk and x0 terms have nonzero coefficient
- CRC checksums are easily implemented in hardware

CRC polynomials

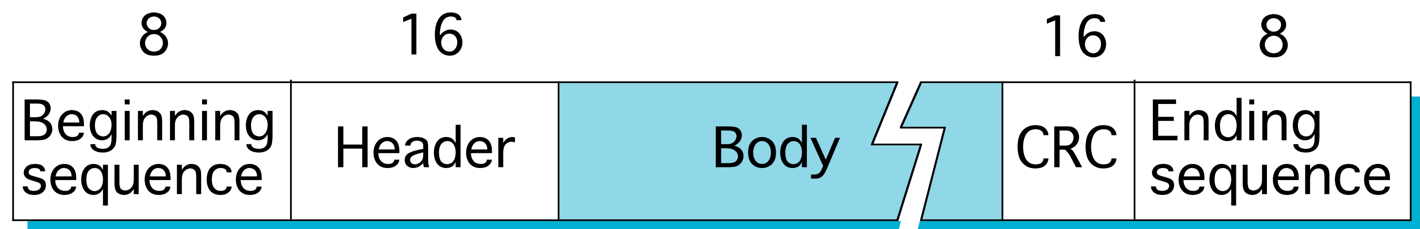
- CRC-8: $x^8+x^2+x^1+1$
- CRC-10: $x^{10}+x^9+x^5+x^4+x^1+1$
- CRC-12: $x^{12}+x^{11}+x^3+x^2+1$
- CRC-16: $x^{16}+x^{15}+x^2+1$
- CRC-CCITT: $x^{16}+x^{12}+x^5+1$
- CRC-32: $x^{32}+x^{26}+x^{23}+x^{22}+ x^{16}+x^{12}+x^{11}+x^{10}+x^8$ $x^7+x^5+x^4+x^2+1$
- Ethernet uses CRC-32
- HDLC: CRC-CCITT
- ATM: CRC-8, CRC-10, and CRC-32

Take away message

- Choosing the right error checking mechanism is a tradeoff between computational complexity and errors that you want to detect
- Multiple layers will do their own error checking, improving error detection

Questions

- What happens if the error detection mechanism did not detect a particular error?
  - Is it possible at all?
- Going back to yesterdays work:

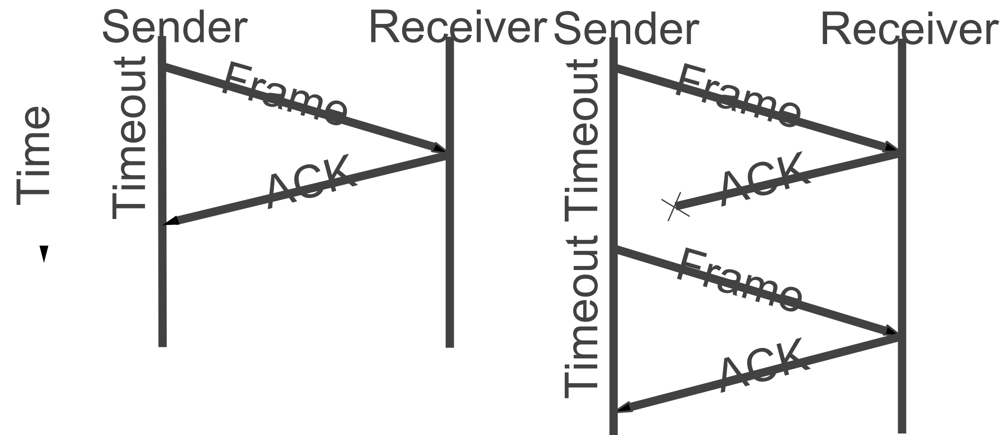


  - What bits should the CRC cover?
  - Should CRC cover the sequence header? Sequence trailer? Why?

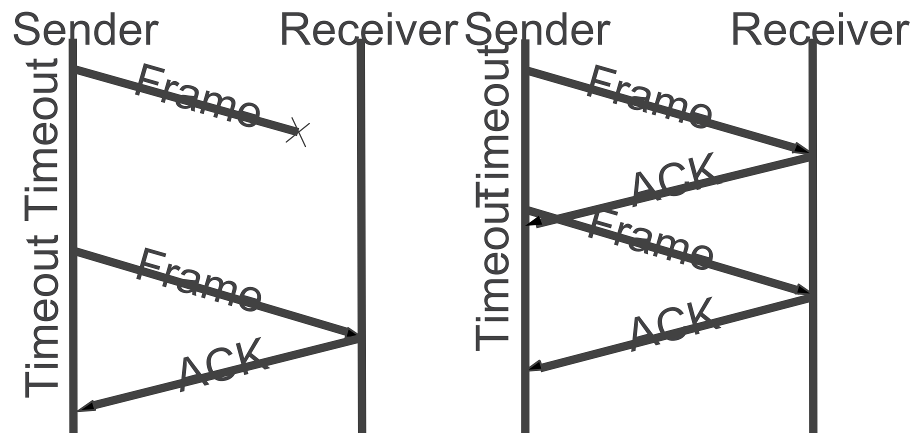
Reliable Transmission

- When corrupt frames are received and discarded, want the network to recover from these errors
- Two fundamental mechanisms:
  - Acknowledgement: A control message sent back to the sender to notify correct receipt
  - Timeout: If sender does not receive and Ack after timeout interval, it should retransmit the original frame
- This general strategy is called ARQ: Automatic Repeat Request
  - Need to select timeout accurately. Too small, we unnecessarily resend frames. Too large, we unnecessarily wait

Acknowledgements & Timeouts



(a) ACK received before timeout (c)

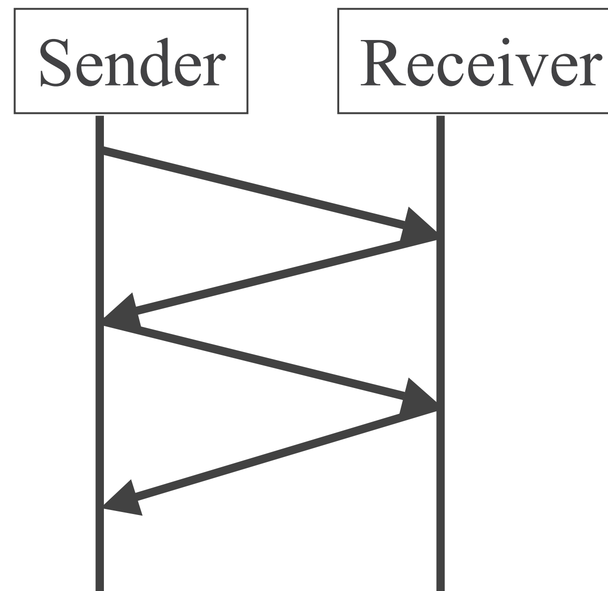


(b) Original frame lost

(d) Premature timeout. Use sequence number fo deal with duplicate frame

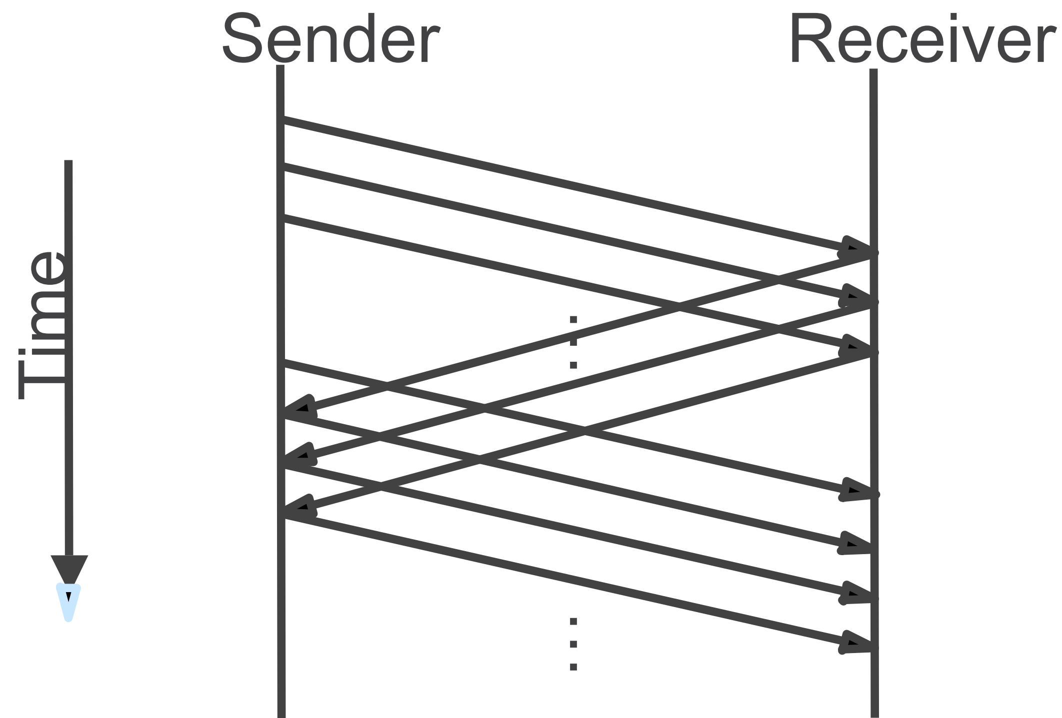
Stop-and-Wait

- Problem: keeping the pipe full
  - (bandwidth delay product)
- Example
  - 1.5Mbps link x 45ms RTT = 67.5Kb (8KB)
  - 1 KB frames imples 1/8th link utilization



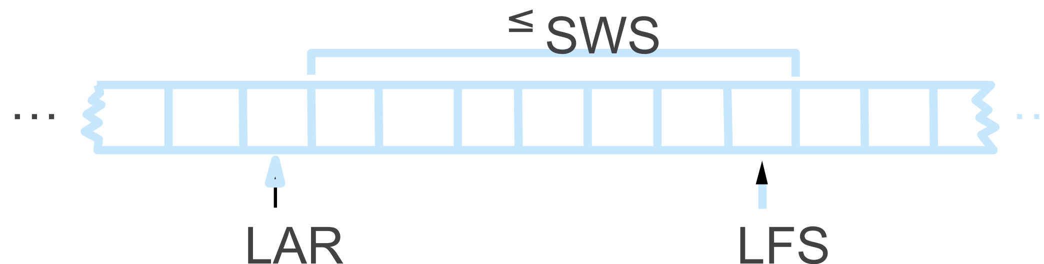
Sliding Window

- Allow multiple outstanding (un-ACKed) frames
- Upper bound on un-ACKed frames, called window



SW: Sender

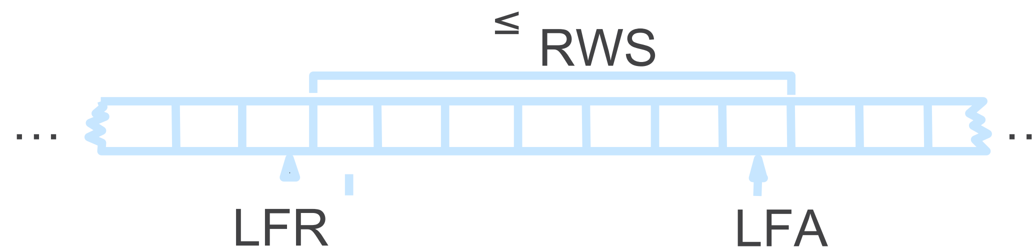
- Assign sequence number to each frame (SeqNum)
- Maintain three state variables:
  - send window size (SWS)
  - last acknowledgment received (LAR)
  - last frame sent (LFS)
- Maintain invariant: LFS - LAR <= SWS



- Advance LAR when ACK arrives
- Buffer up to SWS frames

SW: Receiver

- Maintain three state variables
  - receive window size (RWS)
  - largest frame acceptable (LFA)
  - last frame received (LFR)
- Maintain invariant: LFA - LFR <= RWS



- Frame SeqNum arrives:
  - if LFR < SeqNum < = LFA → accept
  - if SeqNum < = LFR or SeqNum > LFA → discarded
- Send cumulative ACKs

Acknowledgements

- Negative acknowledgment (NAK)
  - When receiver receives frame which has a sequence number higher than the next frame expected, receiver proactively informs the sender to resend the missing frame
- Selective ACK
  - Acknowledge frames that it has received, not just the last frame received

Sequence Number Space

- SeqNum field is finite; sequence numbers wrap around
- Sequence number space must be larger then number of outstanding frames
- SWS <= MaxSeqNum-1 is not sufficient
  - suppose 3-bit SeqNum field (0..7)
  - SWS=RWS=7
  - sender transmit frames 0..6
  - arrive successfully, but ACKs lost
  - sender retransmits 0..6
  - receiver expecting 7, 0..5, but receives second incarnation of 0..5
- SWS < (MaxSeqNum+1)/2 is correct rule
- Intuitively, SeqNum “slides” between two halves of sequence number space

Concurrent Logical Channels

- Multiplex 8 logical channels over a single link
- Run stop-and-wait on each logical channel
- Maintain three state bits per channel
  - channel busy
  - current sequence number out
  - next sequence number in
- Header: 3-bit channel num, 1-bit sequence num
  - 4-bits total
  - same as sliding window protocol
- Separates reliability from order

Take away message

- Reliable delivery means receiver should send acknowledgement.
  - Need to keep timeout just right.
  - Send enough frames to fill pipe (bandwidth delay product)