# Error control

An Engineering Approach to Computer Networking

# CRC

- Detects
  - all single bit errors
  - almost all 2-bit errors
  - any odd number of errors
  - all bursts up to M, where generator length is M
  - longer bursts with probability 2^-m

# Implementation

- Hardware
  - on-the-fly with a shift register
  - easy to implement with ASIC/FPGA
- Software
  - precompute remainders for 16-bit words
  - add remainders to a running sum
  - needs only one lookup per 16-bit block

# Software schemes

- Efficiency is important
  - touch each data byte only once
- CRC
- TCP/UDP/IP
  - all use same scheme
  - treat data bytes as 16-bit integers
  - add with end-around carry
  - one’s complement = checksum
  - catches all 1-bit errors
  - longer errors with prob 1/65536

# Packet errors

- Different from bit errors
  - types
    - not just erasure, but also duplication, insertion,etc.
  - correction
    - retransmission, instead of redundancy

# Types of packet errors

- Loss
  - due to uncorrectable bit errors
  - buffer loss on overflow
    - especially with bursty traffic
      - for the same load, the greater the burstiness, the more the loss
    - loss rate depends on burstiness, load, and buffer size
  - fragmented packets can lead to error multiplication
    - longer the packet, more the loss

# Types of packet errors (cont.)

- Duplication
  - same packet received twice
    - usually due to retransmission
- Insertion
  - packet from some other conversation received
    - header corruption
- Reordering
  - packets received in wrong order
    - usually due to retransmission
    - some routers also reorder

# Packet error detection and correction

- Detection
  - Sequence numbers
  - Timeouts
- Correction
  - Retransmission

# Sequence numbers

- In each header
- Incremented for non-retransmitted packets
- *Sequence space*
  - set of all possible sequence numbers
  - for a 3-bit seq #, space is {0,1,2,3,4,5,6,7}

# Using sequence numbers

- Loss
  - gap in sequence space allows *receiver* to detect loss
    - e.g. received 0,1,2,5,6,7 => lost 3,4
  - acks carry *cumulative* seq #
  - redundant information
  - if no ack for a while, *sender* suspects loss

- Reordering
- Duplication
- Insertion
  - if the received seq # is “very different” from what is expected
    - more on this later

# Sequence number size

- Long enough so that sender does not confuse sequence numbers on acks
- E.g, sending at < 100 packets/sec (R)
  - wait for 200 secs before giving up (T)
  - receiver may dally up to 100 sec (A)
  - packet can live in the network up to 5 minutes (300 s) (*maximum packet lifetime)*
  - can get an ack as late as 900 seconds after packet sent out
  - sent out 900*100 = 90,000 packets
  - if seqence space smaller, then can have confusion
  - so, sequence number > log (90,000), at least 17 bits
- In general 2^seq_size > R(2 MPL + T + A)

# MPL

- How can we bound it?
- Generation time in header
  - too complex!
- Counter in header decremented per hop
  - crufty, but works
  - used in the Internet
  - assumes max. diameter, and a limit on forwarding time

# Sequence number size (cont.)

- If no acks, then size depends on two things
  - reordering span: how much packets can be reordered
    - e.g. span of 128 => seq # > 7 bits
  - burst loss span: how many consecutive pkts. can be lost
    - e.g. possibility of 16 consecutive lost packets => seq # > 4 bits
  - In practice, hope that technology becomes obselete before worst case hits!

# Packet insertion

- Receiver should be able to distinguish packets from other connections
- Why?
  - receive packets on VCI 1
  - connection closes
  - new connection also with VCI 1
  - delayed packet arrives
  - could be accepted
- Solution
  - flush packets on connection clos
  - can’t do this for connectionless networks like the Internet

# Packet insertion in the Internet

- Packets carry source IP, dest IP, *source port number, destination port number*
- How we can have insertion?
  - host A opens connection to B, source port 123, dest port 456
  - transport layer connection terminates
  - new connection opens, A and B assign the same port numbers
  - delayed packet from old connection arrives
  - insertion!

# Solutions

- Per-connection *incarnation number*
  - incremented for each connection from each host
  - - takes up header space
  - - on a crash, we may repeat
    - need stable storage, which is expensive
- Reassign port numbers only after 1 MPL
  - - needs stable storage to survive crash

# Solutions (cont.)

- Assign port numbers serially: new connections have new ports
  - Unix starts at 1024
  - this fails if we wrap around within 1 MPL
  - also fails of computer crashes and we restart with 1024
- Assign initial sequence numbers serially
  - new connections may have same port, but seq # differs
  - fails on a crash
- Wait 1 MPL after boot up (30s to 2 min)
  - this flushes old packets from network
  - used in most Unix systems

# 3-way handshake

- Standard solution, then, is
  - choose port numbers serially
  - choose initial sequence numbers from a clock
  - wait 1 MPL after a crash
- Needs communicating ends to tell each other initial sequence number
- Easiest way is to tell this in a *SYNchronize* packet (TCP) that starts a connection
- 2-way handshake

# 3-way handshake

- Problem really is that SYNs themselves are not protected with sequence numbers
- 3-way handshake protects against delayed SYNs

# Loss detection

- At receiver, from a gap in sequence space
  - send a *nack* to the sender
- At sender, by looking at cumulative acks, and timeing out if no ack for a while
  - need to choose timeout interval

# Nacks

- Sounds good, but does not work well
  - extra load during loss, even though in reverse direction
- If nack is lost, receiver must retransmit it
  - moves timeout problem to receiver
- So we need timeouts anyway

# Timeouts

- Set timer on sending a packet
- If timer goes off, and no ack, resend
- How to choose timeout value?
- Intuition is that we expect a reply in about one round trip time (RTT)

# Timeout schemes

- Static scheme
  - know RTT *a priori*
  - timer set to this value
  - works well when RTT changes little
- Dynamic scheme
  - measure RTT
  - timeout is a function of measured RTTs

# Old TCP scheme

- RTTs are measured periodically
- Smoothed RTT (*srtt*)
- *srtt = a * srtt + (1-a) * RTT*
- timeout = *b * srtt*
- a = 0.9, b = 2
- sensitive to choice of a
  - a = 1 => timeout = 2 * initial srtt
  - a = 0 => no history
- doesn’t work too well in practice

# New TCP scheme (Jacobson)

- introduce new term = mean deviation from mean (m)
- *m = | srtt - RTT |*
- *sm = a * sm + (1-a) * m*
- *timeout = srtt + b * sm*

# Intrinsic problems

- Hard to choose proper timers, even with new TCP scheme
  - What should initial value of srtt be?
  - High variability in R
  - Timeout => loss, delayed ack, or lost ack
    - hard to distinguish
- Lesson: use timeouts rarely

# Retransmissions

- Sender detects loss on timeout
- Which packets to retransmit?
- Need to first understand concept of error control window

# Error control window

- Set of packets sent, but not acked
- 1 2 3 4 5 6 7 8 9 (original window)
- 1 2 3 4 5 6 7 8 9 (recv ack for 3)
- 1 2 3 4 5 6 7 8 9 (send 8)
- May want to restrict max size = window size
- Sender blocked until ack comes back

# Go back N retransmission

- On a timeout, retransmit the entire error control window
- Receiver only accepts in-order packets
- + simple
- + no buffer at receiver
- - can add to congestion
- - wastes bandwidth
- used in TCP
- if packet loss rate is *p*, and

# Selective retransmission

- Somehow find out which packets lost, then only retransmit them
- How to find lost packets?
  - each ack has a bitmap of received packets
    - e.g. cum_ack = 5, bitmap = 101 => received 5 and 7, but not 6
    - wastes header space
  - sender periodically asks receiver for bitmap
  - fast retransmit

# Fast retransmit

- Assume cumulative acks
- If sender sees repeated cumulative acks, packet likely lost
- 1, 2, 3, 4, 5 , 6
- 1, 2, 3      3   3
- Send cumulative_ack + 1 = 4
- Used in TCP

# SMART

- Ack carries cumulative sequence number
- Also sequence number of packet causing ack
- 1 2 3 4 5 6 7
- 1 2 3    3 3 3
- 1 2 3    5 6 7
- Sender creates bitmap
- No need for timers!
- If retransmitted packet lost, periodically check if cumulative ack increased.