Error control

Digital Communications II

Michaelmas Term 2007
Based on Prof. Jon Crowcroft's notes, and thus transitively on
S. Keshav's "An Engineering Approach to Computer Networking"

What you saw in DigiComms I

- What it looks like on the wire
  - Systematic or random noise
  - Attenuation
  - Signal to noise ratio
- Error Detection and Correction
  - Forward Error Correction (FEC)
  - Parity
  - Block codes
  - CRCs

What you saw in DigiComms I (2)

- Protocols
  - Retransmit packets on NACKs or timeouts
- Why single Automatic Repeat Request (ARQ) is poor
  - Want to maximise traffic in transit (within the network)
- Continuous ARQ: using a window
  - Go back N
  - Selective retransmission

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
    - Networks errors: usually burst, not randomly distributed
  - Longer bursts with probability 2^{-m}

Implementation

- Hardware
  - On-the-fly with a shift register
  - Easy to implement with ASIC/FPGA
- Software
  - You saw worked examples in DigiComms I
  - Pre-compute remainders for 16-bit words
  - Add remainders to a running sum
  - Needs only one lookup per 16-bit block

5

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 probability 1/65536

6

Packet errors

- Different from bit errors
  - Types
    - Not just erasure, but also duplication, insertion,etc.
  - Correction
    - Retransmission, instead of redundancy

7

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

8

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
    - May be multiple routing paths

9

Packet error detection and correction

- Detection
  - Sequence numbers
  - Timeouts
- Correction
  - Retransmission

10

Sequence numbers

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

11

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* sequence number
  - Redundant information
  - If no ACK for a while, *sender* suspects loss
- Reordering
- Duplication
- Insertion
  - If the received sequence number is “very different” from what is expected
    - More on this later…

12

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 seconds before giving up (T)
  - Receiver may dally up to 100 seconds (A)
  - Packet can live in the network up to 5 minutes (300s) (*maximum packet lifetime)*
  - Can get an ACK as late as 900 seconds after packet sent
  - Sent out 900*100 = 90,000 packets
  - If sequence space smaller, then can have confusion
  - So, sequence number > log (90,000), at least 17 bits
- In general 2^sequence_size > R(2 MPL + T + A)

Maximum Packet Lifetime (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
    - Time To Live (TTL)
  - Assumes a maximum network 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 → sequence number > 7 bits
  - Burst loss span: how many consecutive packets. can be lost
    - e.g. possibility of 16 consecutive lost packets → sequence number > 4 bits
- In practice, hope that technology becomes obsolete 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 close
  - Can't do this for connectionless networks like the Internet

Packet insertion in the Internet

- TCP/UDP packets carry source IP, destination IP, *source port number, destination port number*
- How we can have insertion?
  - Host A opens connection to B, source port 123, destination port 456
  - Transport layer connection terminates
  - New connection opens, A and B assign the same port numbers
  - Delayed packet from old connection arrives
  - Insertion!

17

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

18

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 sequence # 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

19

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

20

3-way handshake

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

- (Server passive opens a port e.g. 80)
- Client active opens connection with SYN packet
  - Client's sequence number is 'c'
- Server replies with SYN-ACK – ACK=(c+1), SYN=(s)
  - Server's sequence number is 's'
- Client sends ACK to server – ACK=(s+1).

- Recall that TCP sequence numbers are per data byte

21

Loss detection

- At receiver, from a gap in sequence space
  - Send a *NACK* to the sender
- At sender, by looking at cumulative ACKs, and timing out if no ACK for a while
  - Need to choose timeout interval

22

NACKs

- Sounds good, but does not work well
  - Adds extra load during times of loss:
    - Probably not helpful if loss is from congestion!
  - At least NACKs travel in the reverse direction…

- If NACK is lost, receiver must retransmit it
  - Moves timeout problem to receiver

- So we need timeouts anyway

23

Timeouts

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

- Aside: Timeouts usually preclude freeing their related resources
  - What about faked requests?
- SYN attack
  - Forge an unused IP address
  - Send a long sequence of SYN packets to server
    - Up to four minutes' resource allocation per SYN while server retries its SYN-ACKs
  - Without protection, server falls over

24

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

25

Old TCP scheme

- RTTs are measured periodically
- Smoothed RTT (*srtt*) → the exponential moving average
- *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

26

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*

27

Intrinsic problems

- Hard to choose proper timers, even with new TCP scheme
  - What should initial value of *srtt* be?
  - High variability in *RTT*
  - Timeout → loss, delayed ACK, or lost ACK
    - Hard to distinguish
- Lesson: use timeouts rarely

28

Retransmissions

- Sender detects loss on timeout
- Which packets to retransmit?
- We review the concept of the error control window

29

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 (receive 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

30

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 some TCP implementations

31

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

32

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

33

SMART

- ACK carries cumulative sequence number
- Also sequence number of packet causing ACK
  - 1 2 3 4 5 6 7 – packets sent
  - 1 2 3 3 3 3 – received ACK (cumulative)
  - 1 2 3 5 6 7 – received ACK (for packet)
- Sender creates bitmap
- No need for timers!
- If retransmitted packet lost, periodically check if cumulative ACK increased.

34