Today

- shared memory example
- Two party reliable protocols
- Continuing from last time...
- Maximum window size
- Detecting Errors (checksums, CRCs)
- layer 2 vs layer 4
- Bisync
- ARPANET
- Layer 2: Packet boundaries
- Flow control
- round trip delay
- Multiplexing: Protocol Types and Ports
Resources

- Lots of information available as RFCs and Internet Drafts
- Go to www.ietf.org
Two-Way Reliable Protocols

- Used in layer 2 or 4
- Need retransmit timer to avoid deadlock
- Need message numbers, ack numbers
- Pipelining allows multiple outstanding (unack’d) messages
- If receiver discards out of order, then window can be n-1, but not n
- If receiver keeps out of order, window can be n/2, not bigger
Sliding Window Protocol

Assume keep out-of-order packets
Sequence number=3 bits
Assume allowed to send 5 in pipeline

0,1, 2, 3, 4

Acknowledgment (willing to accept 5, 6, 7, 0, 1)

0,1,2,3,4

Acknowledgment
accept 0,1, discard 2-4 as duplicate

5,6,7,0,1

?
Pipelining if accept out of order packets

0, ×, 2, 3

ACK(0)

timeout

1, 2

ACK(3)

4, 5
Detecting Corrupted Data

- (nonmaliciously corrupted data)
- parity (# of 1 bits mod 2)
- VRC (Vertical Redundancy Check): parity on a byte
- LRC (Longitudinal Redundancy Check): parity on a bit on all bytes in a block
- LRC+VRC detects all 1, 2, 3, bit errors, all odd number, some even number
- two’s complement checksum: add all bytes, throw away carries
- one’s complement checksum: add all bytes, add carry over into sum
- CRC: make message divisible by some polynomial
- Error correcting codes
Intuition Behind CRC

- Don’t think about polynomials. Think about numbers
- Message is some number, say 5283
- “CRC polynomial” is some number, say 17
- Multiply 5283 by 100 (shift it over enough places so the fudge thing we’ll compute will fit at the end)
- Divide result by 17, get remainder
- 17 goes into 5283 31076 times, remainder 8
- Subtract 8 from 528300=528292
- Receiver checks for divisibility by 17. If not, error! If yes, either correct, or error is a multiple of 17, hopefully unlikely
- Round to nearest 100=528300
- Truncate 2 digits (divide by 100) get 5283
What’s different about layer 2 than layer 4?

- reordered messages
- variable delay
- bigger round trip delay
- unpredictable packet size
Bisync

- Really old layer 2 protocol. No pipelining. 1 bit sequence number
- Receiver 3 second timeout (send NAK if nothing within 3 seconds)
- Transmitter 2 second timeout (transmit ENQ if nothing within 3 seconds)
- No message number!
- Special characters: ACK0, ACK1, NAK, WACK (Ack but not ready for more), TTD (transmitter delay), ENQ (hey, what’s going on?)
- If special characters appear in the data, can be prefixed with “DLE”
- Can’t retransmit a message except after NAK, or ACK number indicating it didn’t arrive
Bisync

M → ACK0
M → ACK1
M × timeout
ENQ → ACK1
M → ACK0
M × timeout
NAK
ARPANET layer 2 protocol

- 8 bit “sequence number”, not used as sequence number!
- Instead considered 8 separate 1-bit sequence numbers!
- Receiver returns 8-bits, representing sequence number for each “channel”
- Similar to Bisync, but doing 8 flows in parallel
- Compare this to an 8-bit sequence number, or a 3-bit sequence number
Packet Boundaries (Layer 2 issue)

- Layer 1 sometimes sends a stream of bits, sometimes bytes at a time.
- BOB...data....EOB
- What does BOB and EOB look like that can’t occur in the data?
  - header contains byte count (DDCMP)
  - bit stuffing (HDLC)
  - byte stuffing (bisync)
Bit Stuffing

- Flags to delimit packet consist of 6 1’s in a row
- Tell your chip “start packet”, send data stream, tell it to “end packet”
- When tell it to “start” or “end” it puts out a “flag”=01111110
- When in “data” mode, it stuff a 0 after 5 1’s, and if see 5 1’s followed by a 0, it deletes the 0

Data to transmit

111111111011101111100111111000

As transmitted

111110111101110111110001111101000
Byte Stuffing

- In Bisync, BOB and EOB are DLE STX, DLE ETX
- OK for STX and ETX to appear in data, since only the sequence DLE STX and DLE ETX is special
- If DLE appears in data, must add an extra DLE even if next char is not STX or ETX
- Common in other things to have an escape character before other characters to “quote it”
Flow Control

- So Xmitter doesn’t overrun receiver
- Receiver gives “permissions” based on buffers
- Complication: Receiver might give “optimistic” permission based on having a bunch of processes all needing to receive. “Overbooking”. Means can “unpermit” (or close window). But permissions can get lost, reordered... Strategies:
  - (ISO’s layer 4 did this) Ack of ack, to make sure latest ack received. Ack has a “subsequence number” (ACK(i,j) means packet i, but the j’th permission after i)
  - (TCP does this). If receiver hasn’t opened window, Xmitter periodically tries.
  - (DECnet’s layer 4 did this). If transmitter hasn’t sent anything, resend permission
Flow Control

- Flow control
- Usually ACK(i) means I got everything up to i (rather than needing to individually ack each pkt)
- NACK(i) means I lost “i” but don’t retransmit i+1, i+2, etc.
- Novell’s protocol allows retransmit of only missing stuff
  - Sender sends a “burst” with at most same number of pkts as bitmask size
  - Receiver sends a bitmask indicating “missing blocks
  - Transmitter transmits only those
  - Repeat until bitmask indicates all received
**Full Duplex**

- Means data flowing in both directions
- “Piggybacked ack” means in header, put ack number for data from other direction

```
msg 1, ack 0
msg 2, ack 0
msg 3, ack 0
msg 1, ack 2
msg 2, ack 3
msg 4, ack 2
```
TCP (Transmission Control Protocol)

- TCP numbers “bytes”, not “packets”. “Window” permitted by receiver is also bytes.
- It could take 5 packets, and 202 bytes, to send a single character!
  
  - `sending i, ack j, window=1000`
  - ack rcpt
  - `sending j-1, ack i+1, window=0`
    - app took data
    - `sending j-1, ack i+1, window=1`
      - echo character
      - `sending j, ack i+1, window=1`
      - `sending i, ack j+1, window=1000`

- “Nagle’s Alg”: don’t send tiny amt if stuff in pipe. Delay hoping to glom stuff together.
Silly Window Syndrome

- Another pathological behavior of a naive TCP implementation
- Assume transmitter has infinite stuff to send
- Assume receiver has 1000 bytes of buffer

1-980

ack 981, window=20

20 bytes

ack 981, window=1000

ack 1001, window=980

next 980 bytes
Cure for This Sort of Thing

- Delay transmitting if only allowed to send a little bit, if stuff already in pipeline
- Don’t give a small window if your buffer can’t hold a full packet (“segment”) or is more than half full
- So either transmitter or receiver can avoid the problem
- First example: “Nagle’s algorithm” says don’t send a tiny amount of data (less than packet size) if stuff already in pipe, and haven’t already delayed by some amount of time
Timeout Value

- In layer 2 protocols, often a constant, or configured
- In layer 4 protocol, might be very different for different destinations, and might be different if congestion, or different paths
- Typical in layer 4 to measure round trip time, always adapting
- Cute problem: If measure from retransmitted packet, can get fooled into a timer half as big

```
timeout 1
  1  ack 1

timeout 2  ack 1
  2  ack 2
  ack 2
```
Multiplexing

- Multiple processes use the “connection”
- Over layer 2... might be DECnet, IP, routing protocols. Usually long-lived processes
- Over layer 4... usually dynamically created processes. Time sharing system. e.g., 2 users simultaneously want to do a file transfer to the same destination machine
- One mechanism: “protocol type”, a “well-known” value which would be specified someplace... e.g., DECnet would always be 7, Appletalk would be 41...
- Another mechanism: “ports” or “Service Access Points (SAP)”. Two values: one for source, one for destination. Of local significance, but some “well-known”
TCP Multiplexing

- TCP has source, destination port in header
- ("socket"= (IP address, port) pair)
- A process has to say “send me things addressed to this destination port”, as well as what to fill into the source port when sending
- Would be hard if both ports dynamically assigned
- Usually, a protocol has a “well-known” socket that it “listens” at, and other end sends to that well-known socket number, so in TCP header, one port would be “well-known” and other dynamically assigned
TCP Header

- 16-bit source port
- 16-bit destination port
- 32-bit sequence number
- 32-bit ack number
- 4-bit header length
- 6 bits “reserved”
- Flags
  - URG, ACK, PSH, RST, SYN, FIN
- 16-bit window size
- 16-bit TCP checksum
- 16-bit urgent pointer
- options (variable length) (e.g., packet size you’ll accept)
TCP Checksum

- Includes “pseudoheader” as well as TCP header. A “layer violation”
  - IP source address
  - IP destination address
  - “protocol type” in IP header
  - length of this packet
UDP (User Datagram Protocol)

- Header
  - 16 bit source port
  - 16-bit destination port
  - length (header+data)
  - checksum (with same pseudoheader as TCP)
- No retransmissions, message numbers, connect requests, etc.
- Other protocols (e.g., IPX, Appletalk) put the ports in the layer 3 header, so unless you need reliability there wouldn’t be a layer 4 header
- Whether it’s in “layer 3” or “layer 4” is just “specsmanship”