**Reliable Data Transfer**

**Principles of Reliable data transfer**
- Important in app., transport, link layers
- Top-10 list of important networking topics!

**Characteristics of unreliable channel**
- Will determine complexity of reliable data transfer protocol (rdt)

**Reliable data transfer: getting started**
- Incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)
- Consider only unidirectional data transfer
- But control info will flow in both directions!
- Use finite state machines (FSM) to specify sender, receiver

**Rdt1.0: reliable transfer over a reliable channel**
- Underlying channel perfectly reliable
  - No bit errors
  - No loss of packets

**Rdt2.0: channel with bit errors**
- Underlying channel may flip bits in packet
  - Checksum to detect bit errors
  - The question: how to recover from errors:
    - Acknowledgements (ACKs): receiver explicitly tells sender that pkt received OK
    - Negative acknowledgements (NAKs): receiver explicitly tells sender that pkt had errors
    - Sender retransmits pkt on receipt of NAK

- New mechanisms in rdt2.0 (beyond rdt1.0): error detection
- Receiver feedback: control msgs (ACK, NAK) rcvr->sender
rdt2.1: sender, handles garbled ACK/NAKs

rdt2.0: FSM specification

rdt2.0: error scenario

rdt2.0 has a fatal flaw!

- Stop and Wait
  - Sender sends one packet, then waits for receiver response
- What happens if ACK/NAK corrupted?
  - sender doesn’t know what happened at receiver!
  - can’t just retransmit: possible duplicate
- Handling duplicates:
  - sender adds sequence number to each pkt
  - sender retransmits current pkt if ACK/NAK garbled
  - receiver discards (doesn’t deliver up) duplicate pkt

rdt2.1: receiver, handles garbled ACK/NAKs

rdt2.1: receiver, handles garbled ACK/NAKs
rdt2.1: discussion

Sender:
- seq # added to pkt
- two seq. #s (0,1) will suffice. Why?
- must check if received ACK/NAK corrupted
- twice as many states
  - state must "remember" whether "current" pkt has 0 or 1 seq. #

Receiver:
- must check if received packet is duplicate
- state indicates whether 0 or 1 is expected pkt seq #
- note: receiver can not know if its last ACK/NAK received OK at sender

rdt2.2: a NAK-free protocol

- same functionality as rdt2.1, using ACKs only
- instead of NAK, receiver sends ACK for last pkt received OK
- receiver must explicitly include seq # of pkt being ACKed
- duplicate ACK at sender results in same action as NAK: retransmit current pkt

rdt2.2: sender, receiver fragments

New assumption:
underlying channel can also lose packets (data or ACKs)
- checksum, seq. #, ACKs, retransmissions will be of help, but not enough

Approach:
sender waits "reasonable" amount of time for ACK
- retransmits if no ACK received in this time
- if pkt (or ACK) just delayed (not lost):
  - retransmission will be duplicate, but use of seq. #s already handles this
  - receiver must specify seq # of pkt being ACKed
- requires countdown timer

rdt3.0: channels with errors and loss

rdt3.0 sender

rdt3.0 in action
**rdt3.0 in action**

- **Transport Layer**
- **3-19**

**Performance of rdt3.0**

- rdt3.0 works, but performance stinks
- Why?

**rdt3.0: stop-and-wait operation**

- **Transport Layer**
- **3-21**

**Pipelined protocols**

- Pipelining: sender allows multiple, "in-flight", yet-to-be-acknowledged packets
- range of sequence numbers must be increased
- buffering at sender and/or receiver
- Two generic forms of pipelined protocols: go-Back-N, selective repeat

**Pipelining: increased utilization**

- **Transport Layer**
- **3-23**

**Go-Back-N**

- Sender:
  - k-bit seq # in pkt header
  - "window" of up to N, consecutive unack'd pkts allowed
  - ACKs all pkts up to, including seq # n - "cumulative ACK"
  - may deceive duplicate ACKs (see receiver)
  - timer for each in-flight pkt
  - timeout(n): retransmit pkt n and all higher seq # pkts in window
GBN: receiver

- **ACK-only:** always send ACK for correctly-received pkt with highest *in-order* seq #
  - may generate duplicate ACKs
  - need only remember expected seq num
- **out-of-order pkt:**
  - discard (don’t buffer) → isn’t this bad?
  - Re-ACK pkt with highest in-order seq #
  - may generate duplicate ACKs
  - need only remember expected seq num

**Selective Repeat**

- **receiver** individually acknowledges all correctly received pkts
  - buffers pkts, as needed, for eventual in-order delivery to upper layer
- **sender** only resends pkts for which ACK not received
  - sender timer for each unACKed pkt
- **sender window**
  - N consecutive seq #s
  - again limits seq #s of sent, unACKed pkts

**Selective repeat in action**

Sender:
- data from above:
  - if next available seq # in window, send pkt
- timeout(n):
  - resend pkt n, restart timer
- ACK(n) in (sendbase, sendbase+N):
  - send ACK(n)
  - out-of-order: buffer
  - in-order: deliver (also deliver buffered, in-order pkts), advance window to next not-yet-received pkt
- pkt n in (recvbase, recvbase+1):
  - ACK(n)
- otherwise:
  - ignore

Receiver:
- pkt n in (recvbase, recvbase+1):
  - send packet n
  - send ACK(n)
  - out-of-order: buffer
  - in-order: deliver (also deliver buffered, in-order pkts), advance window to next not-yet-received pkt
- pkt n in (recvbase, recvbase+1):
  - ACK(n)
  - otherwise:
  - ignore

**Selective repeat: sender, receiver windows**

Sender view of sequence numbers
- window size:
- (a) sendbase
- (b) recvbase
- already acked
- sent, not yet sent
- not sent
- not usable

Receiver view of sequence numbers
- window size:
- (a) sendbase
- (b) recvbase
- out of order
- in order
- usable
- not usable

**Selective repeat in action**

- sender pkt 0, send pkt 1
- receiver pkt 0, send ACK(0)
- sender pkt 1, send pkt 2
- receiver pkt 1, send ACK(1)
- sender pkt 2, send ACK(2)
- receiver pkt 2, send ACK(2)
- sender pkt 3, send ACK(3)
- receiver pkt 3, send ACK(3)
- sender pkt 4, send ACK(4)
- receiver pkt 4, send ACK(4)
- sender pkt 5, send ACK(5)
- receiver pkt 5, send ACK(5)

Transport Layer 3-25

Transport Layer 3-26

Transport Layer 3-27

Transport Layer 3-28

Transport Layer 3-29

Transport Layer 3-30
Selective repeat: dilemma

Example:
- seq #s: 0, 1, 2, 3
- window size: 3
- receiver sees no difference in two scenarios!
- incorrectly passes duplicate data as new in (a)

Q: what relationship between seq # size and window size?