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

We'll:
- incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)
- consider only unidirectional data transfer
  - but control info will flow on 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
- separate FSMs for sender, receiver:
  - sender sends data into underlying channel
  - receiver reads data from underlying channel
Rdt2.0: channel with bit errors

- underlying channel may flip bits in packet
  - recall: UDP 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
  - human scenarios using ACKs, NAKs?
- new mechanisms in rdt2.0 (beyond rdt1.0):
  - error detection
  - receiver feedback: control msgs (ACK, NAK) rcvr->sender

rdt2.0: FSM specification

sender FSM

receiver FSM

rdt2.0: in action (no errors)

sender FSM

receiver FSM

rdt2.0: in action (error scenario)

sender FSM

receiver FSM
**rdt2.0 has a fatal flaw!**

What happens if ACK/NAK corrupted?
- sender doesn’t know what happened at receiver!
- can’t just retransmit: possible duplicate

What to do?
- sender ACKs/NAKs receiver’s ACK/NAK? What if sender ACK/NAK lost?
- retransmit, but this might cause retransmission of correctly received pkt!

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

**stop and wait**
Sender sends one packet, then waits for receiver response

---

**rdt2.1: sender, handles garbled ACK/NAKs**

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 NAKs 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

**rdt3.0: channels with errors and loss**

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

**Q:** How to deal with loss?
- Sender waits until certain data or ACK lost, then retransmits
- Yuck: drawbacks?

**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 sender**

**rdt3.0 in action**

(a) Operation with no loss

(b) Lost packet
**Performance of rdt3.0**

- rdt3.0 works, but performance stinks
- example: 1 Gbps link, 15 ms e-e prop. delay, 1KB packet:
  
  $T_{transmit} = \frac{8\text{kb/pkt}}{10^{9}\text{ b/sec}} = 8 \text{ microsec}$

  Utilization $U = \frac{\text{fraction of time sender busy sending}}{30.016 \text{ msec}} = \frac{8 \text{ microsec}}{\text{0.00015}}$

  - 1KB pkt every 30 msec -> 33kB/sec throuput over 1 Gbps link
  - network protocol limits use of physical resources!

**Pipelined protocols**

Pipelining: sender allows multiple, “in-flight”, yet-to-be-acknowledged pkts

- range of sequence numbers must be increased
- buffering at sender and/or receiver

- Two generic forms of pipelined protocols: **go-Back-N**, selective repeat

**Go-Back-N**

**Sender:**
- k-bit seq # in pkt header
- “window” of up to N, consecutive unack’d pkts allowed

- ACK(n): 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: sender extended FSM

GBN: receiver extended FSM

receiver simple:
- ACK-only: always send ACK for correctly-received pkt with highest *in-order seq #*
  - may generate duplicate ACKs
  - need only remember *expectedseqnum*
- out-of-order pkt:
  - discard (don’t buffer) → *no receiver buffering!*
  - ACK pkt with highest in-order seq #

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: sender, receiver windows

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]:
    - Mark pkt n as received
    - If n smallest unACKed pkt, advance window base to next unACKed seq #

Receiver:
- Pkt n in [rcvbase, rcvbase+N-1]
- Send ACK(n)
- Out-of-order: buffer
- In-order: deliver (also deliver buffered, in-order pkts), advance window to next not-yet-received pkt

Selective repeat in action

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?