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

![Diagram of transport layer with provided service and service implementation](image)

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 in both directions!
- Use finite state machines (FSM) to specify sender, receiver

![Diagram of FSM states and events](image)

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

![Diagram of rdt_send and rdt_rcv](image)
**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) rcv→sender

**Rdt2.0: FSM specification**

**Rdt2.0: operation with no errors**

**Rdt2.0: error scenario**

**Rdt2.0 has a fatal flaw!**

**Handling duplicates:**
- sender retransmits current pkt if ACK/NAK garbled
- sender adds sequence number to each pkt
- receiver discards (doesn't deliver up) duplicate pkt

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

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

**Rdt2.1: sender, handles garbled ACK/NAKs**
rdt2.1: receiver, 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 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

rdt3.0 sender

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

Approach: sender waits “reasonable” amount of
- 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
Performance of rdt3.0

- rdt3.0 works, but performance stinks
- example: 1 Gbps link, 15 ms e-e prop. delay, 1KB packet:
  \[ T_{\text{transmit}} = \frac{L}{R} \text{ (packet length in bits) } \]
  \[ = \frac{10^{12} \text{ bps} \times 10^{-6}}{10^{9} \text{ b/sec} } = 8 \text{ microsec} \]
- \[ U_{\text{sender}} = \frac{L/R}{RTT + L/R} = \frac{0.008}{0.0008} = 0.0027 \]
- 1KB pkt every 30 msec = 33Kbps throughput 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 receive duplicate ACKs (see receiver)
- timer for each in-flight pkt
- timeout(n): retransmit pkt $n$ and all higher seq # pkts in window

GBN: receiver extended FSM

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!
  - Re-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 retransmits 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

window size

N

already
ack-ed
sent, not
yet ack-ed
usable, not
yet sent
not usable
Selective repeat

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 unACKd pkt, advance window base to next unACKed seq #

receiver
- pkt n in [robase,robase+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 [robase-H,robase-I):
    - ACK(n)
  - otherwise:
    - ignore

Selective repeat: dilemma

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

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