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)

**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
- the question: how to recover from errors:
  - acknowledgements (ACKs): receiver explicitly tells sender that packet was received correctly
  - negative acknowledgements (NAKs): receiver explicitly tells sender that packet had errors
  - sender retransmits packet 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) rcv→sender

**Reliable data transfer: getting started**

- **send side**
  - `rdt_send()`: called from above, (e.g., by app.). Passed data to deliver to receiver upper layer (receiver app)
  - `udt_send()`: called by rdt, to transfer packet over unreliable channel to receiver

- **receive side**
  - `rdt_rcv()`: called when packet arrives on rcv-side of channel
  - `deliver_data()`: called by rdt to deliver data to upper layer

**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
- use finite state machines (FSM) to specify sender, receiver

**state**: From current “state” next state uniquely determined by next event

- event: causing state transition
- actions taken on state transition

- **unreliable channel**

**Rdt1.0: 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
- the question: how to recover from errors:
  - acknowledgements (ACKs): receiver explicitly tells sender that packet was received correctly
  - negative acknowledgements (NAKs): receiver explicitly tells sender that packet had errors
  - sender retransmits packet 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) rcv→sender
rdt2.0: FSM specification

rdt2.0: in action (no errors)

rdt2.0: in action (error scenario)

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

rdt2.1: receiver, handles garbled ACK/NAKs

Application Layer
**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

**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 data or ACK lost, then retransmits
- How do you know when the data is lost?

**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 in action**

(a) operation with no loss

(b) lost packet

(c) lost ACK

(d) premature timeout
Performance of rdt3.0

- rdt3.0 works, but performance stinks
- example: 1 Gbps link, 15 ms e-e prop. delay, 1KB packet:
  \[ T_{\text{transmit}} = \frac{8 \text{kbit/pkt}}{10^9 \text{b/sec}} = 8 \text{ microsec} \]
  \[ \text{Utilization} = U = \frac{8 \text{ microsec}}{30.016 \text{ msec}} = 0.00015 \]
  - 1KB pkt every 30 msec -> 33kB/sec throughput over 1 Gbps link
  - network protocol limits use of physical resources!

Pipelined protocols

- Go-Back-N
  - \( k \)-bit seq # in pkt header
  - “window” of up to \( N \), consecutive unack’ed 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
  - \( \text{timeout}(n) \): retransmit pkt \( n \) and all higher seq # pkts in window

- GBN: sender extended FSM
  - \( n \): get seq num from \( \text{rcv pkts} \)
  - if unexpected seq
    - \( \text{timeout}(n) \)
  - if seq in window
    - \( \text{ack}(n) \)

- GBN in action
  - 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) -> no receiver buffering!
    - ACK pkt with highest in-order seq #
Problems with GBN

- Retransmits entire sender window on timeout
  - Can cause excessive retransmissions
  - Problem is exacerbated for networks with large "memory", i.e. large delay bandwidth product
- Receiver throws away any out of order packets, even if they are received correctly.
  - Forces retransmission

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

Selective repeat in action
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?
Out of Order Delivery

- What happens if the network delivers packets out of order
  - Send order ≠ receive order
- Need a much larger – potentially infinite - sequence space
  - Why?