Reliable Data Transfer

Recap: Reliable data transfer
- Goal: provide reliable packet delivery over unreliable channels.
- We have considered bit flips as the only channel impairment.
- Proposed a stop-and-wait protocol based on ACK/NACK.

rdt2.0 has a fatal flaw!
- Handling duplicates:
  - Sender retransmits current pkt if ACK/NACK garbled.
  - Sender adds sequence number to each pkt.
  - Receiver discards (doesn't deliver up) duplicate pkt.

rdt2.1: sender, 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.
  - Receiver:
    - Must check if received packet is duplicate.
    - State indicates whether 0 or 1 is expected pkt.
    - Note: Receiver cannot know if its last ACK/NACK 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 sender

- start_timer
- timeout
- rdt_rcv(rcvpkt)
- stop_timer
- sender FSM

rdt3.0 in action

(a) operation with no loss
(b) lost packet
(c) lost ACK
(d) premature timeout

rdt3.0 in action

(a) operation with no loss
(b) lost packet
(c) lost ACK
(d) premature timeout

rdt3.0 in action

(a) operation with no loss
(b) lost packet
(c) lost ACK
(d) premature timeout

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 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 # already handles this
    - receiver must specify seq # of pkt being ACKed
  - requires countdown timer

rdt2.2: sender, receiver fragments

- sender FSM
- receiver FSM

rdt3.0: sender

- start_timer
- timeout
- rdt_rcv(rcvpkt)
- stop_timer
- sender FSM

rdt3.0 in action

(a) operation with no loss
(b) lost packet
(c) lost ACK
(d) premature timeout

rdt3.0 in action

(a) operation with no loss
(b) lost packet
(c) lost ACK
(d) premature timeout

rdt3.0 in action

(a) operation with no loss
(b) lost packet
(c) lost ACK
(d) premature timeout

rdt3.0 in action

(a) operation with no loss
(b) lost packet
(c) lost ACK
(d) premature timeout

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:
  \[ T_{\text{transmit}} = \frac{L}{R} \text{ (packet length in bits)} \]
  \[ \frac{8\text{kb/pkt}}{10^9 \text{ b/sec}} = 8 \text{ microsec} \]
- \( U_{\text{sender}} \): utilization - fraction of time sender busy sending
  \[ U_{\text{sender}} = \frac{L}{R} \left( \frac{\text{RTT} + L}{R} \right) = 0.00027 \]
- 1KB pkt every 30 msec \( \rightarrow \) 33kB/sec thruput over 1 Gbps link
- network protocol limits use of physical resources!

Pipelining: increased utilization

- Increase utilization by a factor of 3!

Go-Back-N

- 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

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 extended FSM

- Flowchart and code snippet for GBN protocol

Transport Layer
GBN: receiver

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

Selective Repeat

- **GBN** is somewhat wasteful, as correctly received but out-of-order packets are sent again.
- **In 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

Window Size

- The window size limits actual throughput.
- Why bother with having a window at all?
- It is used to do flow control and congestion control.
- This will be discussed next week.

Selective repeat: sender, receiver windows

![Diagram showing 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 \([\text{sendbase}, \text{sendbase}+N-1]\):
  - mark pkt n as received
  - if n smallest unACKed pkt, advance window base to next unACKed seq #
- pkt n in \([\text{rcvbase}, \text{rcvbase}+N-1]\): out-of-order pkt, advance window to next not-yet-received pkt

Receiver

- pkt n in \([\text{rcvbase}+N-1, \text{rcvbase}+N]\):
  - deliver (also deliver buffered, in-order pkts)
- otherwise:
  - ignore
**Selective repeat in action**

```
seq #s: 0, 1, 2, 3
window size=3
```

- Receiver sees no difference in two scenarios!
- Incorrectly passes duplicate data as new in (a)

**Example:**

```
/boxshadowup 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?

---

**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?

---

**Forward Erasure/Error Correction: A Different Approach to RDT**

- Our approach to reliable data delivery is based on ACKs and retransmissions, i.e. feedback.
- Long RTTs => long delays and/or low throughput
- An alternative approach is via forward corrections for errors and losses.

---

**Example: Fountain Codes**