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!**

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

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

---

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

- rdt_send(data)
  - sndpkt = make_pkt(0, data, checksum)
  - udt_send(sndpkt)

- rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt)

- Wait for call 0 from above

- rdt_send(data)
  - sndpkt = make_pkt(1, data, checksum)
  - udt_send(sndpkt)

- rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt)

- Wait for ACK or NAK 1

- rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isNAK(rcvpkt)

- Wait for call 1 from above
rdt2.1: receiver, handles garbled ACK/NAKs

```
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && has_seq0(rcvpkt)
extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) && not corrupt(rcvpkt) && has_seq1(rcvpkt)
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)
```

```
Wait for 0 from below
```

```
Wait for 1 from below
```

```
rdt_rcv(rcvpkt) && (corrupt(rcvpkt)
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)
```

```
rdt_rcv(rcvpkt) && (corrupt(rcvpkt) && has_seq0(rcvpkt))
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)
```

Transport Layer 5

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

Transport Layer 6
**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**

```
rdt Send(data)
    sndpkt = make_pkt(0, data, checksum)
    udt_send(sndpkt)

rdt_rcv(rcvpkt) &&
    (corrupt(rcvpkt) ||
    isACK(rcvpkt, 1))
    udt_send(sndpkt)

rdt_rcv(rcvpkt) &&
    notcorrupt(rcvpkt)
    && isACK(rcvpkt, 0)
```

```
Wait for
0 from above
```

Sender FSM

```
rdt_rcv(rcvpkt) &&
    (corrupt(rcvpkt) ||
    has_seq1(rcvpkt))
    udt_send(sndpkt)

extract(rcvpkt, data)
    deliver_data(data)
    sndpkt = make_pkt(ACK1, checksum)
    udt_send(sndpkt)
```

```
Wait for
0 from below
```

Receiver FSM

```
Lambda
```
**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. #'s already handles this
  - receiver must specify seq # of pkt being ACKed
- requires countdown timer

```
rdt3.0 sender
```

```
rdf3.0: sender
```
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:

\[ T_{\text{transmit}} = \frac{L}{R} \] (packet length in bits) / (transmission rate, bps) = \( \frac{8\text{kb/pkt}}{10^{22} \text{ b/sec}} \) = 8 microsec

- \[ U_{\text{sender}}: \text{utilization} \] - fraction of time sender busy sending

\[ U_{\text{sender}} = \frac{L / R}{\text{RTT} + L / R} = \frac{0.008}{30.008} = 0.00027 \]

- 1KB pkt every 30 msec \( \rightarrow \) 33kB/sec thruput over 1 Gbps link
- network protocol limits use of physical resources!

rdt3.0: stop-and-wait operation

\[ \frac{U_{\text{sender}} = \frac{L / R}{\text{RTT} + L / R}}{\text{RTT + L / R}} = \frac{0.008}{30.008} = 0.00027 \]
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

Pipelining: increased utilization

\[
U_{\text{sender}} = \frac{3 \times L / R}{RTT + L / R} = \frac{0.024}{30.008} = 0.0008
\]

Increase utilization by a factor of 3!
Go-Back-N

Sender:
- 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
- timeout(n): retransmit pkt n and all higher seq # pkts in window

GBN: sender extended FSM

```
rdt_send(data)
if (nextseqnum < base+N) {
  sndpkt[nextseqnum] = make_pkt(nextseqnum,data,chksum)
  udt_send(sndpkt[nextseqnum])
  if (base == nextseqnum)
    start_timer
  nextseqnum++
} else
  refuse_data(data)

else
  rdt_rcv(rcvpkt)
  & corrupt(rcvpkt)
  if (base == nextseqnum)
    stop_timer
  else
    start_timer
```

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 #

**GBN in action**

![Diagram of GBN in action](image)
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

- 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
**Selective repeat: sender, receiver windows**

![Diagram of sender and receiver sequence numbers]

**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 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
- pkt n in [rcvbase-N,rcvbase-1]:
  - ACK(n)
- otherwise:
  - ignore
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?
**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**