Performance of rdt3.0

- rdt3.0 works, but performance stinks
- example: 1 Gbps link, 15 ms e-e prop. delay, 1KB packet:
  \[ T_{transmit} = \frac{8\text{kb/pkt}}{10^8 \text{b/sec}} = 8 \text{ microsec} \]
  \[ \text{Utilization} = U = \frac{\text{fraction of time sender busy sending}}{30.016 \text{ msec}} = 0.00015 \]
  - 1KB pkt every 30 msec -> 33kB/sec thruput 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

How much data in the pipeline?

- Q: How many unacknowledged packets?
- A: \( R \times RTT \) - the so-called bandwidth-delay product
- Units: \( R \) is bits/second, RTT is seconds, bandwidth-delay product is bits. Scale by packetsize to get packets.
- Q: but what if packets are different sizes?
- A: deferred to discussion of TCP

Go-Back-N

Sender:
- k-bit seq # in pkt header (finite; requires modular arithmetic)
- “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: sender extended FSM

```c
void rdt_send(data) { 
if (nextseqnum < base+1) { 
    compute checksum; 
    make_pkt(sendpkt[nextseqnum], nextseqnum, data, checksum); 
    udt_send(sendpkt[nextseqnum]); 
} 
if (base == nextseqnum) 
    start_timer; 
nextseqnum = nextseqnum + 1; 
else 
    refuse_data(data); 
}
```

GBN: receiver extended FSM

```c
default 
udt_send(sendpkt) 
extract(rcvpkt, data) 
deliver_data(data) 
make_pkt(sendpkt, ACK, expectedseqnum); 
udt_send(sendpkt) 
```

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 #

GBN in action

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

- 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)
- Similar problem occurs in GBN

Q: what relationship between seq # size and window size?
How big does the seq. no. field have to be to prevent confusion and allow 100% link utilization?

- 100% utilization requires \( R \times RTT / \text{packetsize} \) outstanding packets.
- To prevent confusion we need a sequence number space containing \( 2 \times (R \times RTT / \text{packetsize}) \) values.
- The sequence number field must have \( 1 + \text{ceil}(\log_2(R \times RTT / \text{packetsize})) \) bits.