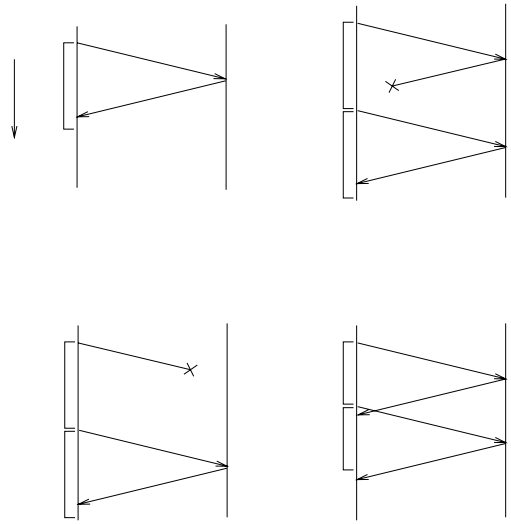
Performance of Stop-and-Wait

Reliable Transmission

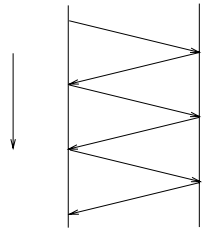
Recover from corrupted and discarded frames

- Error Correcting Codes (ECC) — Forward Error Correction (FEC) ⟵ not good enough
- **Acknowledgements** (ACK) and **Timeouts** — **Automatic Repeat reQuest (ARQ)**



Stop-and-Wait

- After tx'ing one frame, the sender waits for an ACK before tx'ing the next frame
- If ACK didn't arrive after a certain period of time, the sender times out and retx'es the original frame



Problem – *duplicates (lost ACKs or premature timeout)*
Solution – **1-bit sequence #** (since a frame can only be confused with the frame before it or the one after it)

Drawback – *low link utilization*
Solution – **keep the pipe full**

Example – 1.5Mbps link × 45ms RTT = 67.5Kb (≈ 8KB). Assuming frame size of 1KB, stop-and-wait uses about $\frac{1}{8}$ of the link's capacity $\Longrightarrow$ want the sender to be able to transmit up to 8 frames before having to wait for an ACK

Performance of Stop-and-Wait – No Errors

- Consideration transmission in one direction only
- Define

  F = length of frame (in bits)
  D = length of data (info) field (in bits)
  A = length of ACK (in bits)
  C = link capacity (in bits/sec)
  τ = one-way propagation delay & processing time (in sec)
  U = **(Link) Utilization** = fraction of time that *useful data* is being successfully tramsmitted

$$= \frac{time\ to\ tx\ data}{total\ time\ to\ tx\ a\ frame}$$

$$= \frac{D/C}{F/C + \tau + A/C + \tau}$$

Performance of Stop-and-Wait – With Errors

Define

T = Timeout interval
P_1 = probability a data frame is damaged/lost
P_2 = probability an ACK frame is damaged/lost
L = Prob. a data frame or its ACK is damaged/lost
$1 - L =$
so $L =$

Time to *successfully* transmit a frame

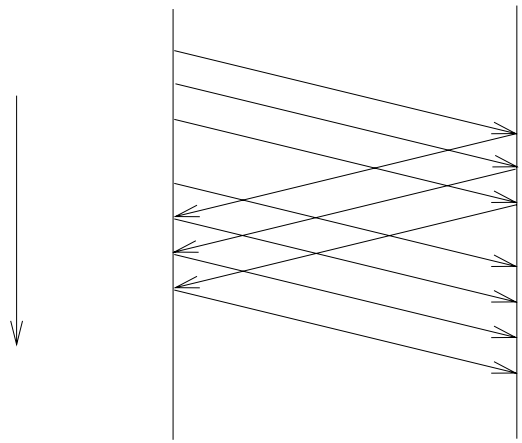
$$= [F/C + 2\tau + A/C] + (F/C + T) * L + (F/C + T) * L^2 + (F/C + T) * L^3 + \cdots$$
$$= [F/C + 2\tau + A/C] + (F/C + T) * \frac{L}{1-L}$$

$$U = \frac{D/C}{F/C + 2\tau + A/C + (F/C + T) * L/(1 - L)}$$

Sliding Window Protocols

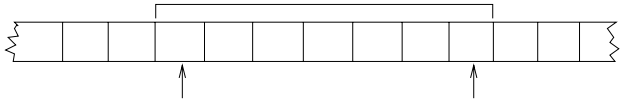
Idea – Allow sender to transmit *multiple* frames before receiving an ACK $\Longrightarrow$ *keeping the pipe full* $\Longrightarrow$ **pipelining**

Example – Assume D×BW = 8KB and frame size = 1KB, we would like the sender to be ready to tx the *9th* frame at about the same time that the ACK for the 1st frame arrives



Sender:

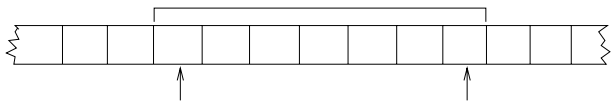
- Assign sequence number to each frame (`SeqNum`)
- Maintain 3 state variables and 1 invariant
  - sending window size (`SWS`) – # of unACKed frames
  - last acknowledgment received (`LAR`)
  - last frame sent (`LFS`)
  - invariant: `LFS - LAR` $\leq$ `SWS`



- When ACK arrives, advance `LAR` $\rightarrow$ slide (advance) window
- Associate a timer with each outstanding frame
- Retx the frame should the timer expire before an ACK is received
- Buffer up to `SWS` frames for (potential) retransmission

Receiver:

- Maintain 3 state variables and 1 invariant
  - receiving window size (`RWS`) – # of out-of-order frames
  - last frame acceptable (`LFA`)
  - next frame expected (`NFE`)
  - invariant: `LFA - NFE + 1` $\leq$ `RWS`



- Frame `SeqNum` arrives –
  - if (`SeqNum` $<$ `NFE`) or (`SeqNum` $>$ `LFA`) $\Longrightarrow$ discarded
  - if (`NFE` $\leq$ `SeqNum` $\leq$ `LFA`) $\Longrightarrow$ accept

Problems –

- **errors** (damaged/lost frames)
- **finite sequence #**
- **whether to send ACK if an out-of-order frame is received ?**
- solutions – **go-back-N** and **selective repeat**

Go-Back-N

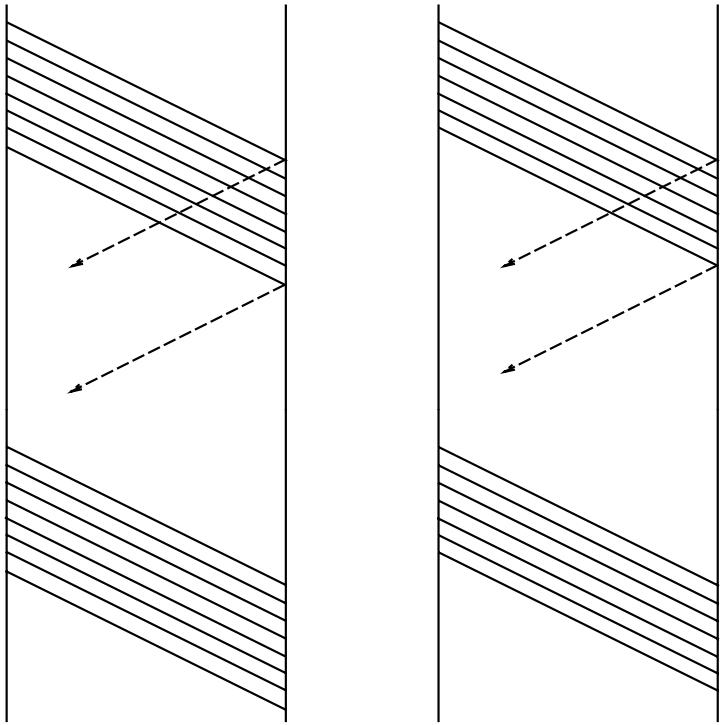
- *Finite* sequence numbers: 0 1 2 3 4 $\cdots$ M
- Maximum sending window size (SWS = w) – maximum # of frames *outstanding* (not yet ACKed)
- **Receiving window size (RWS) = 1**
  - R discards all subsequent frames and sends no ACKs for them
  - S retransmits all unACKed frames starting with the damaged/lost one
- Example – SWS (w) = 3 and $M = 7$

  0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 $\cdots$

  - send 0, 1, 2
  - send 3 only after ACK 0 received
  - send 4 only after ACK 1 received
  - $\cdots$
- Example – SWS (w) = $M + 1$
  - S sends 0 1 2 $\cdots$ M
  - S gets ACK0 ACK1 ACK2 $\cdots$ ACKM
  - S sends **another incarnation** 0 1 2 $\cdots$ M
  - Question - Did R acknowledge new frames or resend old ACKs ???

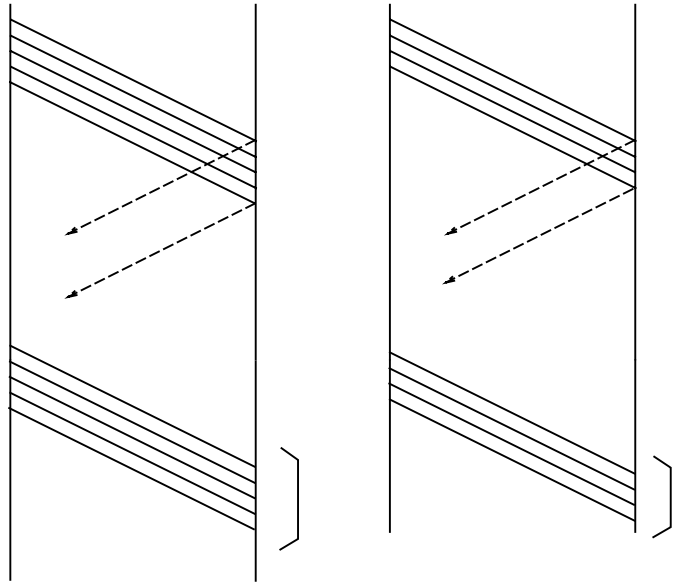
Go-Back-N – SWS and Sequence

- Must have $\boxed{w \leq M}$ to avoid *overlap*



Selective Repeat

- Receiver accepts any frame in its *receiving window* even it's out of order
- Receiving Window Size (RWS) $\equiv 1 \longrightarrow$ Go-Back-N
- Sequence numbers: 0, 1, 2, 3, 4, $\cdots$, M
- Must have $\boxed{w \leq (M+1)/2}$ to avoid *overlap*



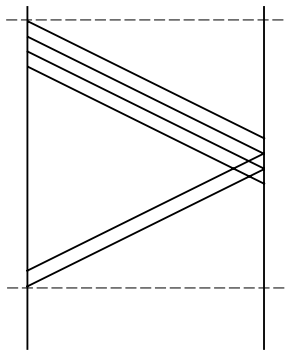
Performance of Go-Back-N

Case 1 – No errors and window *large enough* so we don't have to wait for ACKs

- Define
  - w = Maximum Window Size
  - F = length of frame (in bits)
  - D = length of data (info) field (in bits)
  - C = link capacity (in bits/sec)
  - τ = one-way propagation delay (in sec)
  - wF/C is the time to tx a *full* window
  - 1st frame takes $F/C+\tau$ to reach receiver
  - With a piggybacked ACK, ACK returns after $2F/C+2\tau$
  - *Window large enough* $\Longrightarrow wF/C \geq 2F/C+2\tau$
  - No overhead due to Go-Back-N, except the header
  - $U = \frac{D}{F}$

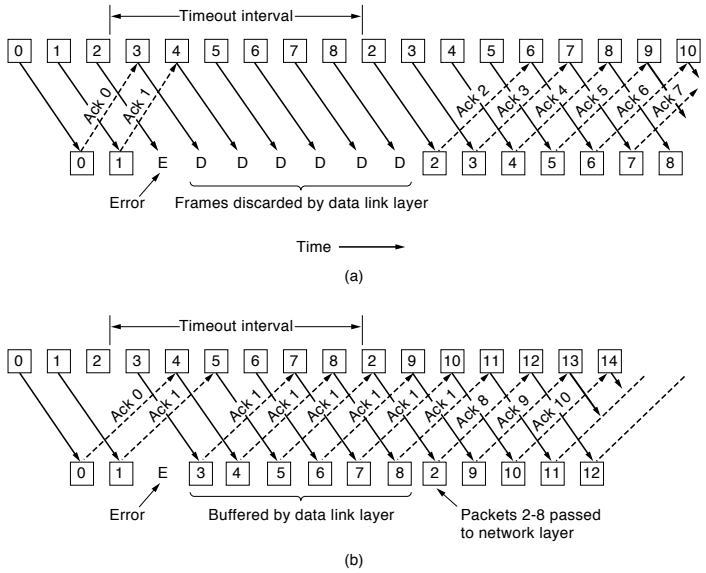
Case 2 – No errors and *small* window to wait for ACKs

- Send w frames, then wait for ACKs
- $U = \frac{wD/C}{2F/C+2\tau} = \frac{wD}{2F+2\tau C}$



Case 3 – With errors (Oh! No...)

Go-Back-N vs. Selective Repeat



Trade-offs

- **bandwidth →**
- **buffer space →**