

Lecture 5:

Reliable Transmission

& Error Handling Wrapup

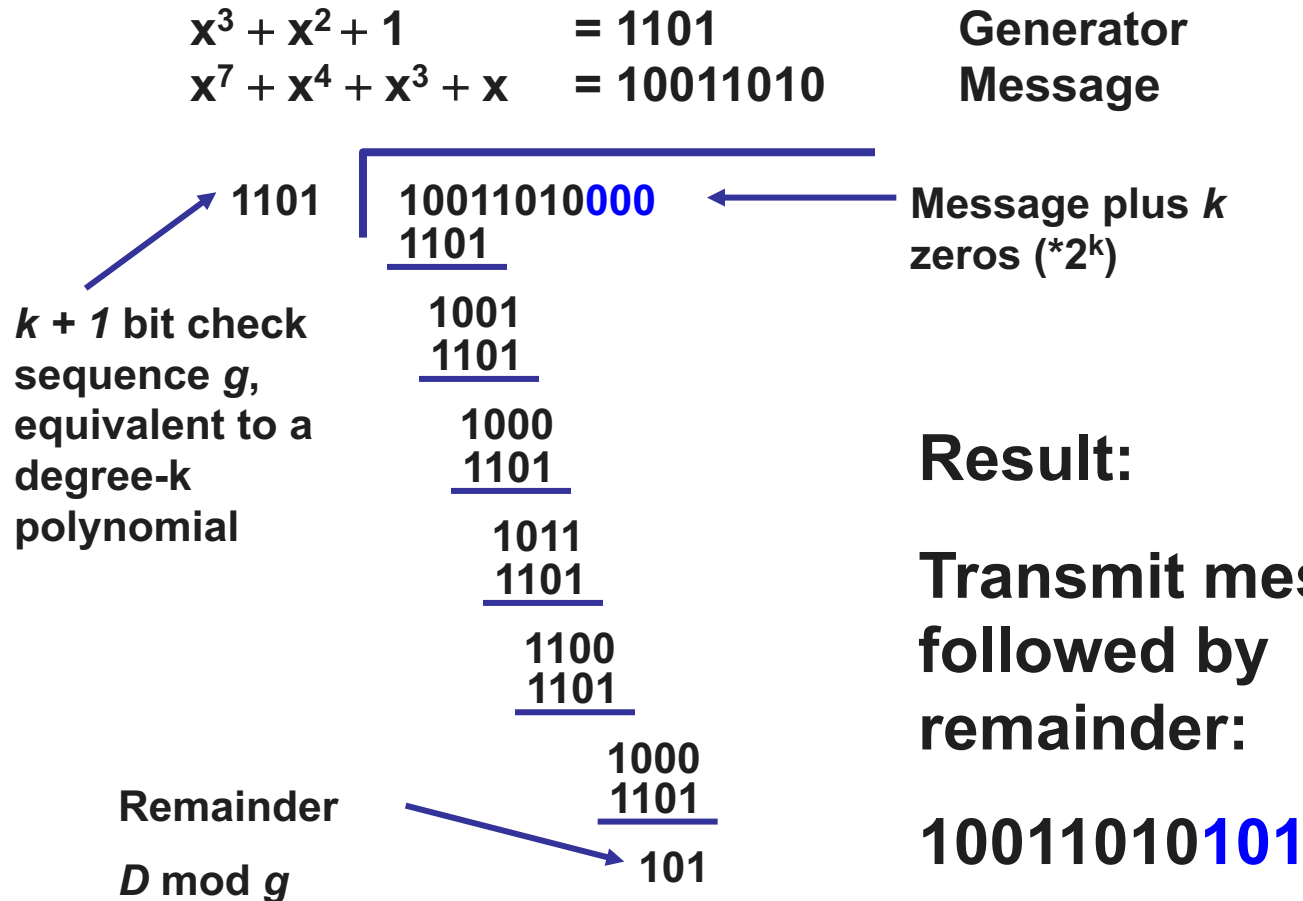
CSE 123: Computer Networks

Aaron Schulman





CRC Example Encoding





CRC Example Decoding

$$\begin{array}{ll} x^3 + x^2 + 1 & = 1101 \\ x^{10} + x^7 + x^6 + x^4 + x^2 + 1 & = 10011010101 \end{array}$$

Generator Received Message

$k + 1$ bit check sequence g , equivalent to a degree- k polynomial

Received message, no errors

Result:

CRC test is passed

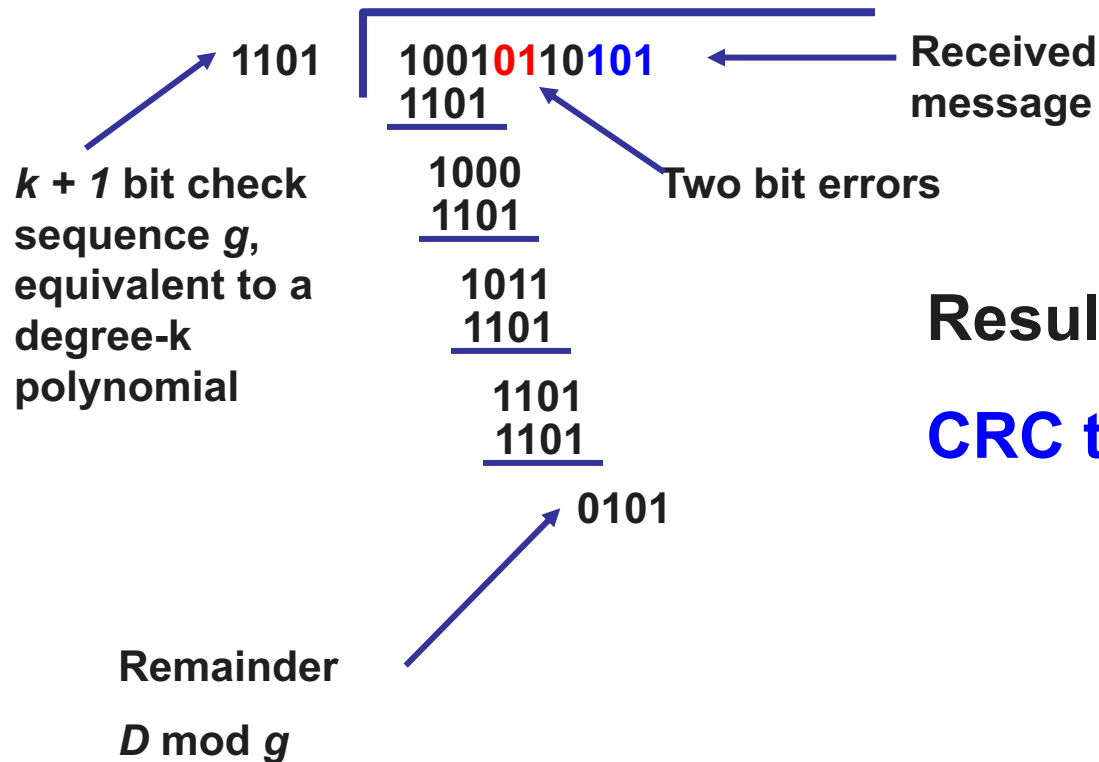
Remainder $D \bmod g$

$$\begin{array}{r} 1101 \overline{) 10011010101} \\ \underline{1101} \\ 1001 \\ \underline{1101} \\ 1000 \\ \underline{1101} \\ 1011 \\ \underline{1101} \\ 1100 \\ \underline{1101} \\ 1101 \\ \underline{1101} \\ 0 \end{array}$$



CRC Example Failure

$x^3 + x^2 + 1 = 1101$ Generator
 $x^{10} + x^7 + x^5 + x^4 + x^2 + 1 = 10010110101$ Received Message



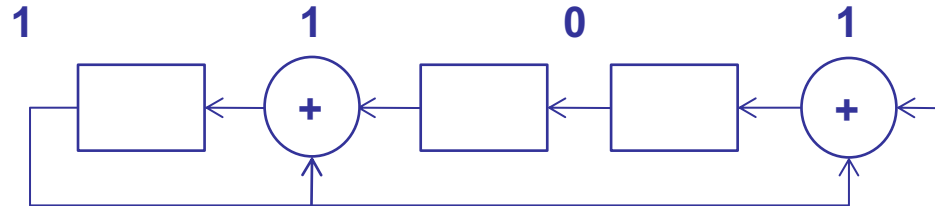


Common Generators

CRC-8	$x^8 + x^2 + x^1 + 1$
CRC-10	$x^{10} + x^9 + x^5 + x^4 + x^1 + 1$
CRC-12	$x^{12} + x^{11} + x^3 + x^2 + x^1 + 1$
CRC-16	$x^{16} + x^{15} + x^2 + 1$
CRC-CCITT	$x^{16} + x^{12} + x^5 + 1$
CRC-32	$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x^1 + 1$



CRC in Hardware



- Key observation is only subtract when MSB is one
 - ◆ Recall that subtraction is XOR
 - ◆ No explicit check for leading one by using as input to XOR
- Hardware cost very similar to most basic checksum
 - ◆ We're only interested in remainder at the end
 - ◆ Not a divider: Only need k registers as remainder is only k bits



Error Handling Summary

- Add redundant bits to detect if frame has errors
 - ◆ A few bits can detect errors
 - ◆ Need more to *correct* errors
- Strength of code depends on Hamming Distance
 - ◆ Number of bitflips between codewords
- Checksums and CRCs are typical methods
 - ◆ Both cheap and easy to implement in hardware
 - ◆ CRC much more robust against burst errors

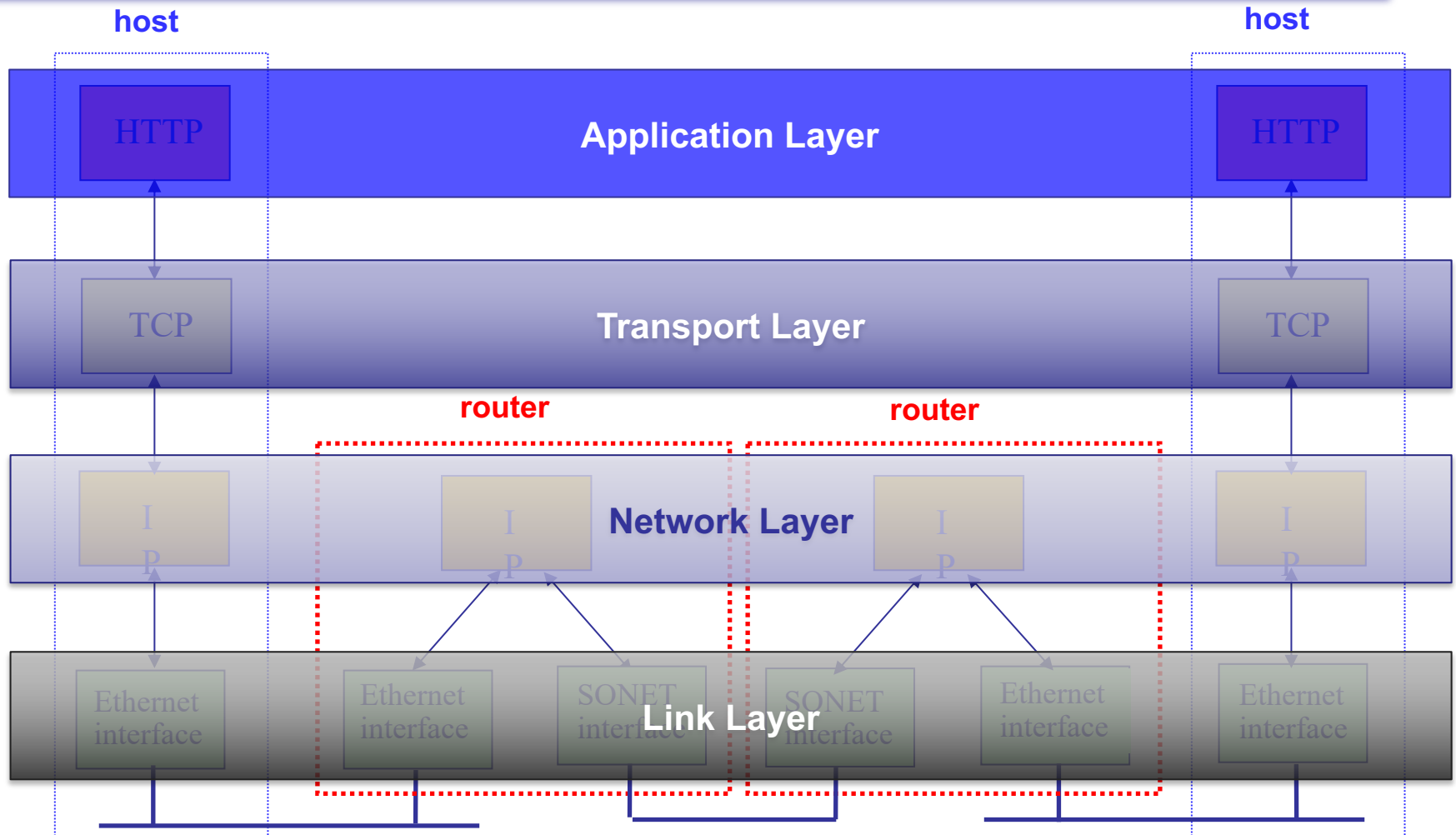


Picking up the Pieces

- Link layer is lossy
 - ◆ We deliberately throw away corrupt frames!
 - ◆ Infrequent bit errors still lead to occasional frame errors
 - » 10,000+ bits in each frame
- Things get even harrier if we consider crossing multiple links
 - ◆ In a few lectures, we'll start sending frames on long trips
 - ◆ Each intermediate link might lose, corrupt, *reorder*, etc.
 - ◆ Regardless of cause, we'll call loss events **drops**
- We want to provide reliable, in-order delivery
 - ◆ Can—and will—do this at multiple layers



Moving up the Stack



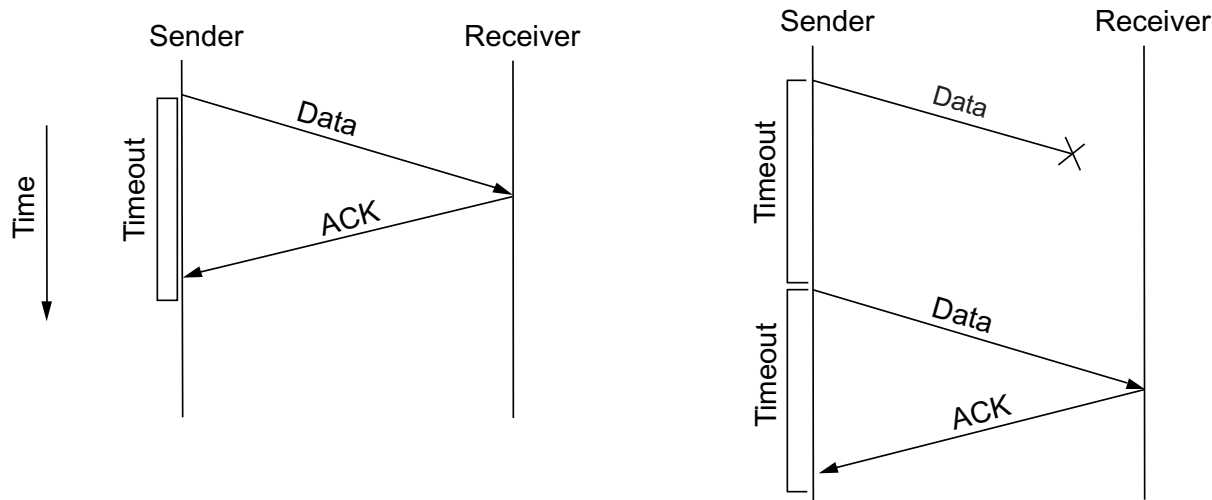


Reliable Transmission

- The **packet**-based version of the same problem
 - ◆ How do we reliably send a message when **packets** (not just bits) can be lost/corrupted in the network?
- Two options
 - ◆ **Detect a loss/corruption and retransmit**
 - ◆ Send data redundantly to tolerate loss/corruption



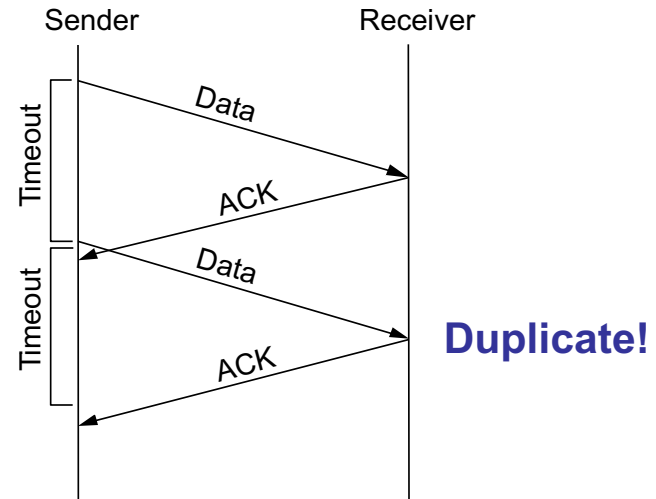
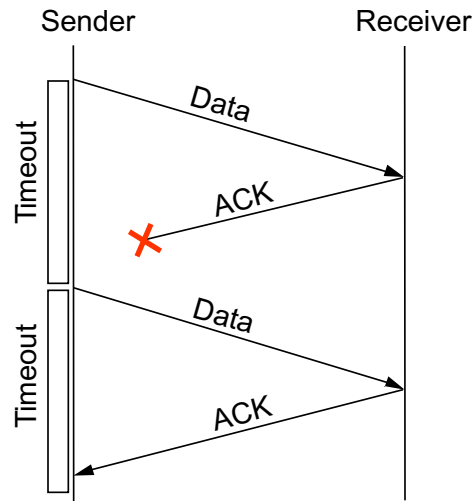
Simple Idea: ARQ



- Receiver sends **acknowledgments** (ACKs)
 - ◆ Sender “times out” and retransmits if it doesn’t receive them
- Basic approach is generically referred to as **Automatic Repeat Request** (ARQ)



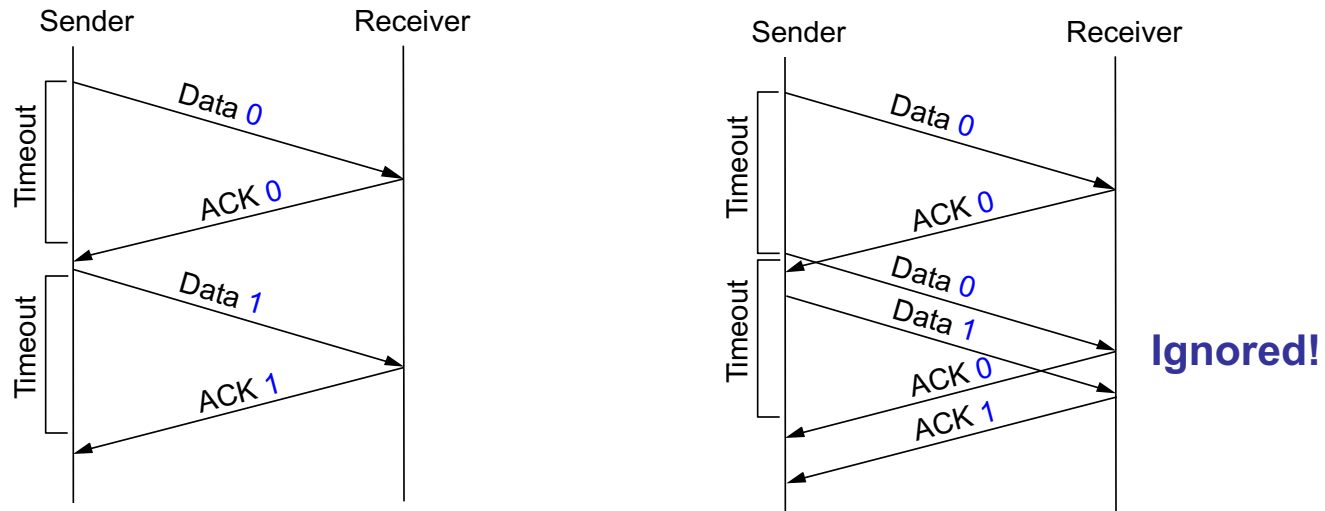
Not So Fast...



- Loss can occur on ACK channel as well
 - ◆ Sender cannot distinguish data loss from ACK loss
 - ◆ Sender will retransmit the data frame
- ACK loss—or early timeout—results in **duplication**
 - ◆ The receiver thinks the retransmission is new data!

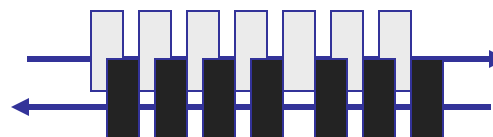
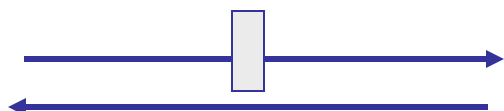


Sequence Numbers



- Sequence numbers solve this problem
 - ◆ Receiver can simply ignore duplicate data
 - ◆ But must still send an ACK! (Why?)
- Simplest ARQ: **Stop-and-wait**
 - ◆ Only allow one outstanding frame at a time

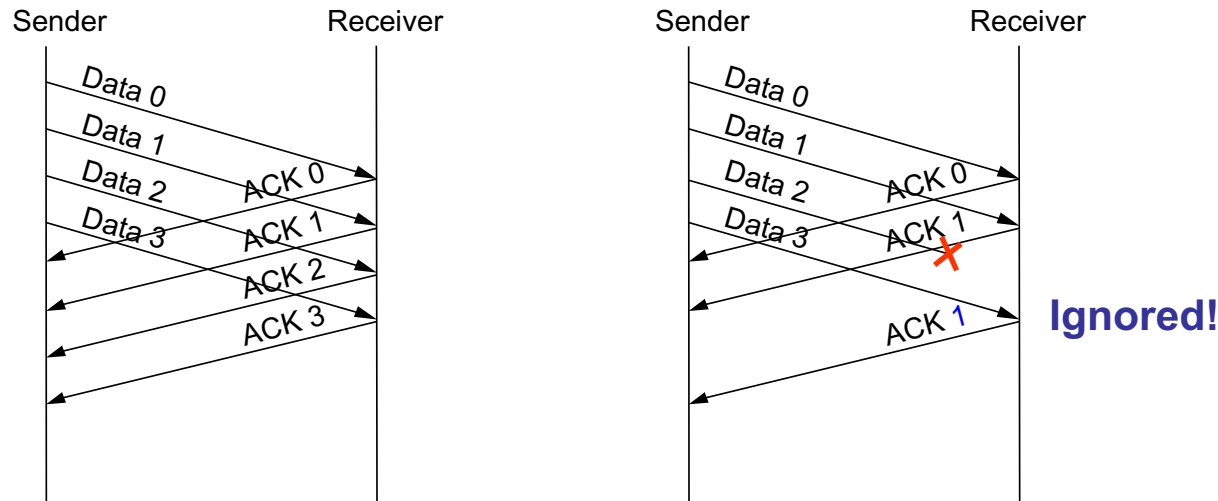
Stop-and-Wait Performance



- **Lousy performance** if $\text{time}(\text{xmit 1 pkt}) \ll \text{propagation delay}$
 - ◆ How bad?
- Want to utilize all available bandwidth
 - ◆ Need to keep more data “in flight”
 - ◆ How much? Called the **bandwidth-delay product**
- Also limited by quality of timeout (how long?)

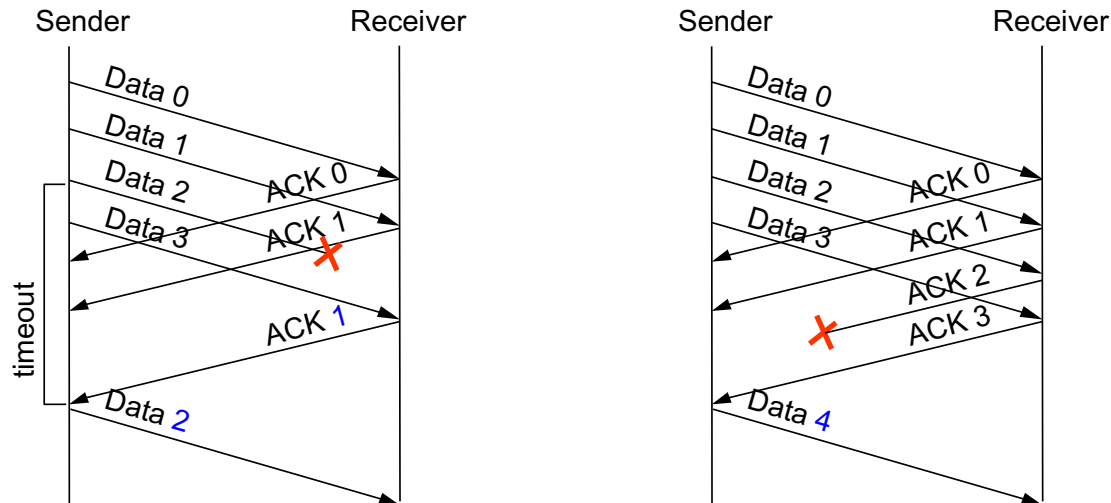


Pipelined Transmission



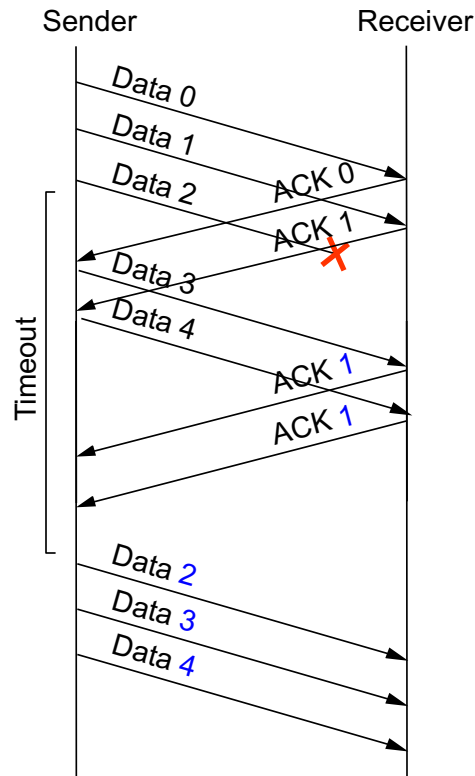
- Keep multiple packets “in flight”
 - ◆ Allows sender to make efficient use of the link
 - ◆ Sequence numbers ensure receiver can distinguish frames
- Sender buffers outstanding un-acked packets
 - ◆ Receiver ACKs the highest *consecutive* frame received
 - » ACKs are **cumulative** (covers current frame and all previous)

Go-Back-N



- Retransmit all packets from point of loss
 - ◆ Packets sent after lost packet (or ACK) are ignored (i.e., sent again)
- Simple to implement
(receiver only has to track one packet at a time)
- Sender controls how much data is “in flight”

Send Window (sender buffer)



**Go-Back-N Example
with window size 3**

- Bound on number of outstanding packets
 - ◆ Window “opens” upon receipt of *new* ACK
 - ◆ Window resets entirely upon a timeout
- Limits amount of memory waste
 - ◆ Don’t need to remember all packets ever sent for retrans.
 - ◆ Go-Back-N might still lead to sending lots of duplicates
 - » We can do better with **selective retransmission**:
only retransmit missed packets



For Next Time

- Read 5-5.1 in P&D
- Homework out Friday and due in 1 week
- (Keep) going on the project...