

Lecture 6:

Reliable Transmission

CSE 123: Computer Networks
Alex Snoeren (guest lecture)





Lecture 6 Overview

- Finishing Error Detection
 - ◆ Cyclic Remainder Check (CRC)
- Handling errors
 - ◆ Automatic Repeat Request (ARQ)
 - ◆ Acknowledgements (ACKs) and timeouts
 - ◆ Stop-and-Wait



From Sums to Remainders

- Checksums are easy to compute, but very fragile
 - ◆ In particular, **burst** errors are frequently undetected
 - ◆ We'd rather have a scheme that “smears” parity
- Need to remain easy to implement in hardware
 - ◆ So far just shift registers and an XOR gate
- We'll stick to Modulo-2 arithmetic
 - ◆ Multiplication and division are XOR-based as well
 - ◆ Let's do some examples...



Modulo-2 Arithmetic

- Multiplication

$$\begin{array}{r} 1101 \\ 110 \\ \hline 0000 \\ 11010 \\ 110100 \\ \hline 101110 \end{array}$$

- Division

$$\begin{array}{r} 1101 \\ 110 \overline{) 101110} \\ \underline{110} \\ 111 \\ \underline{110} \\ 011 \\ \underline{000} \\ 110 \end{array}$$



Cyclic Remainder Check

- Idea is to *divide* the incoming data, D , rather than add
 - ◆ The divisor is called the **generator**, g
- We can make a CRC resilient to k -bit burst errors
 - ◆ Need a generator of $k+1$ bits
- Divide $2^k D$ by g to get remainder, r
 - ◆ Remainder is called **frame check sequence**
- Send $2^k D - r$ (i.e., $2^k D \text{ XOR } r$)
 - ◆ Note $2^k D$ is just D shifted left k bits
 - ◆ Remainder must be at most k bits
- Receiver checks that $(2^k D - r)/g = 0$



Error Detection – CRC

- View data bits, **D**, as a binary number
- Choose $r+1$ bit pattern (generator), **G**
- Goal: choose r CRC bits, **R**, such that
 - ♦ $\langle D, R \rangle$ exactly divisible by G (modulo 2)
 - ♦ Receiver knows G , divides $\langle D, R \rangle$ by G . If non-zero remainder: error detected!
 - ♦ Can detect all burst errors less than $r+1$ bits
- Widely used in practice (Ethernet, FDDI, ATM)

← d bits → ← r bits →

D : data bits to be sent	R : CRC bits
----------------------------	----------------

bit pattern

$D * 2^r \text{ XOR } R$ *mathematical formula*

CRC: Rooted in Polynomials



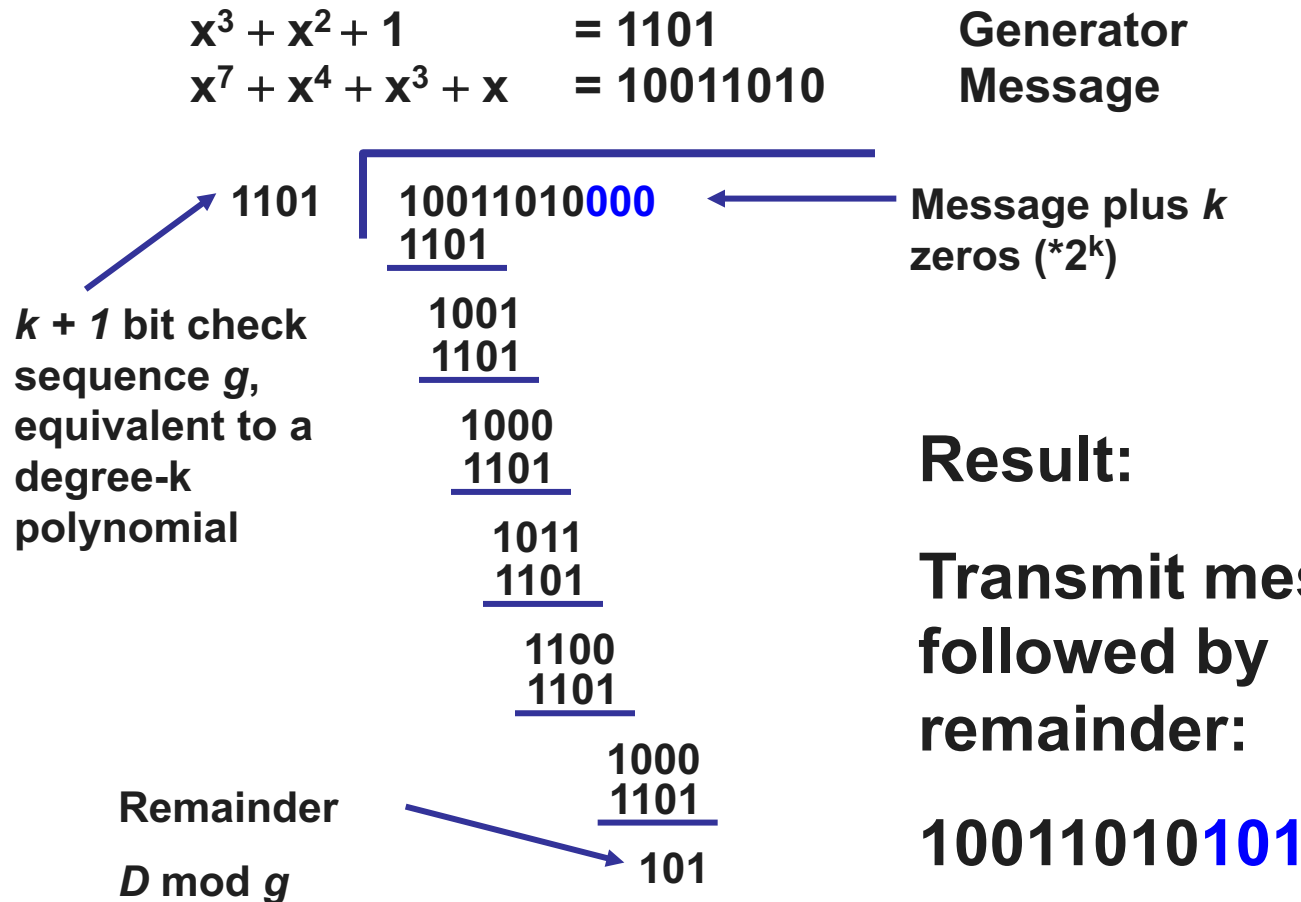
- We're *actually* doing polynomial arithmetic
 - ◆ Each bit is actually a coefficient of corresponding term in a k^{th} -degree polynomial

$$1101 \text{ is } (1 * X^3) + (1 * X^2) + (0 * X^1) + (1 * X^0)$$

- Why do we care?
 - ◆ Can use the properties of finite fields to analyze effectiveness
 - ◆ Says any generator with two terms catches single bit errors

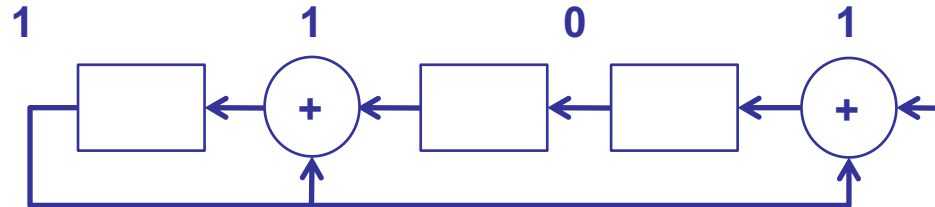


CRC Example Encoding





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 checksum
 - ◆ We're only interested in remainder at the end
 - ◆ Only need k registers as remainder is only k bits



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

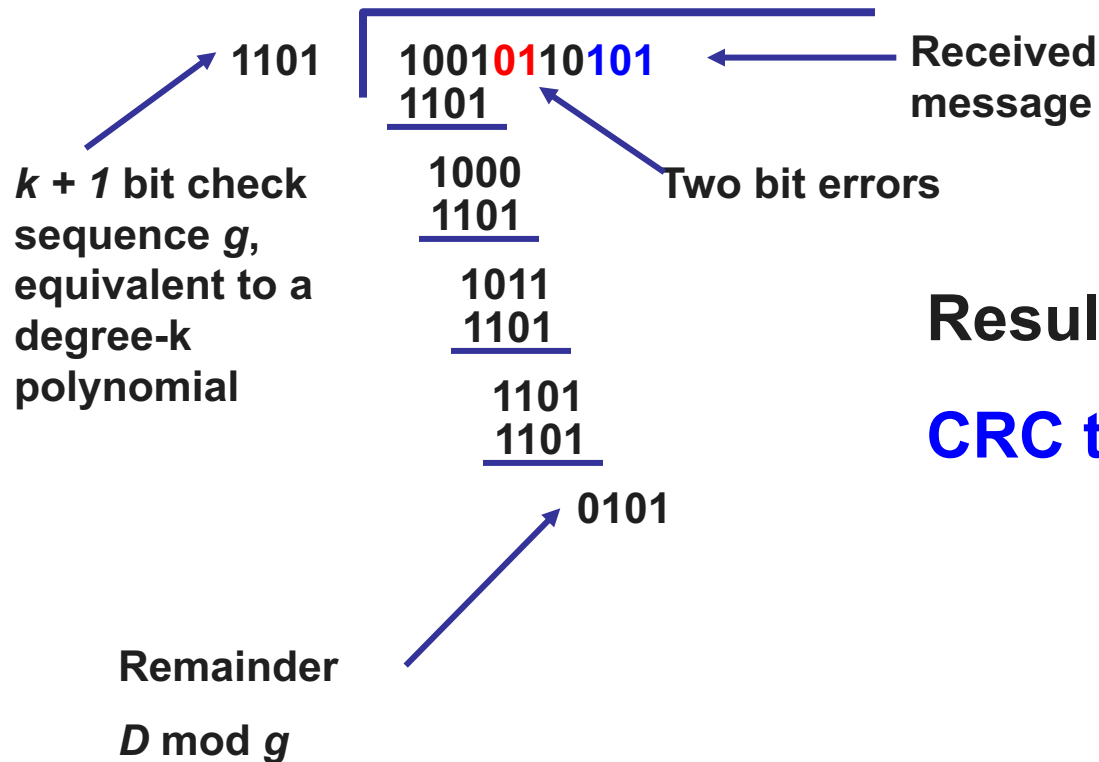
Remainder $D \bmod g$

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



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	$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x^1 + 1$



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

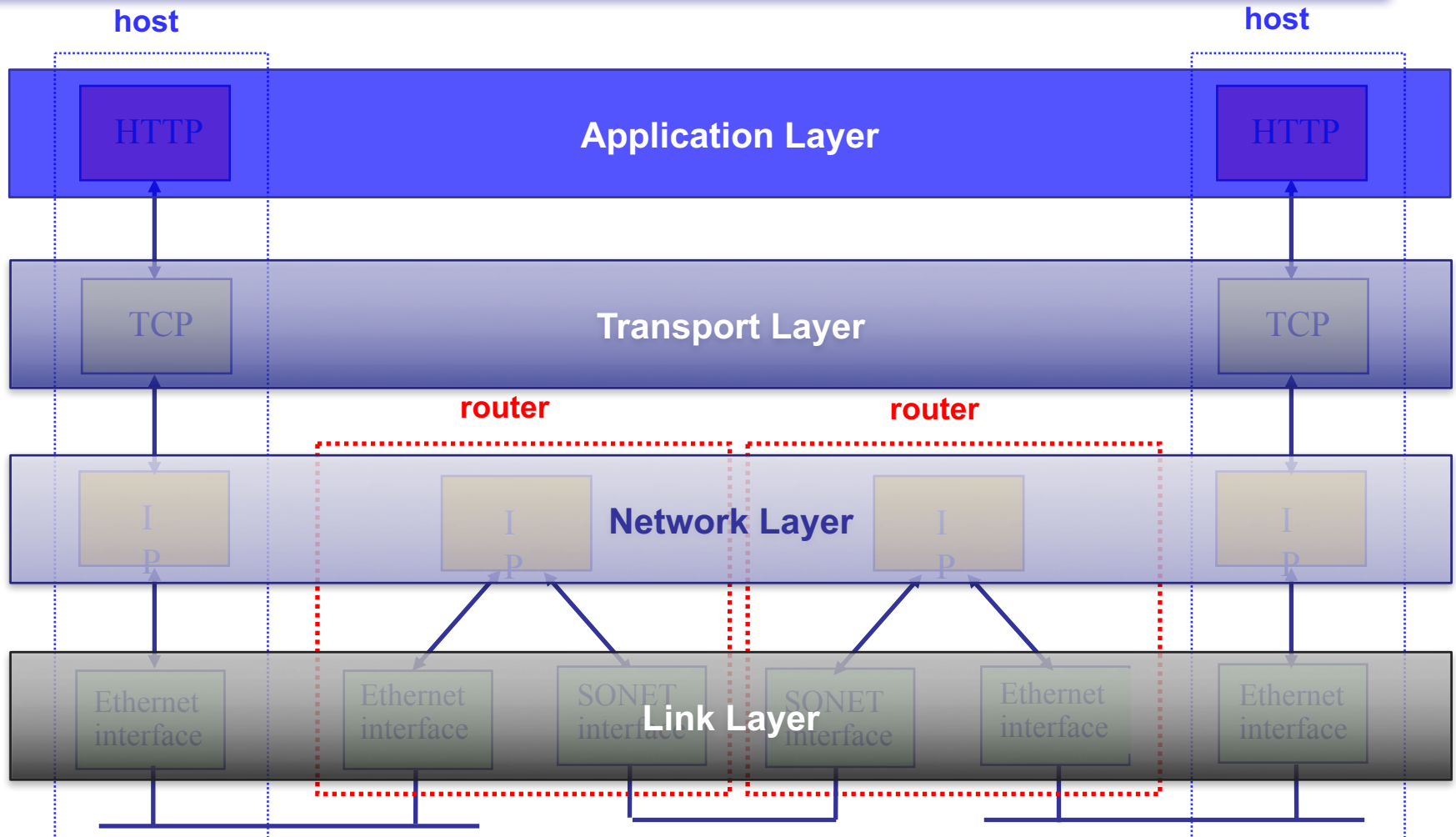


Picking up the Pieces

- Link layer is lossy
 - ◆ We deliberately threw away corrupt frames last lecture
 - ◆ Infrequent bit errors still lead to occasional frame errors
 - » 10,000+ bits in each frame
- Things get even harrier if we consider multiple links
 - ◆ In a few lectures, we'll start sending frames on long trips
 - ◆ Each intermediate stop 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



Stepping back: A thought experiment



- You want to send a long letter to your friend
 - ◆ The only medium available to either of you is *postcards*
 - ◆ Postcards get lost in the mail, delayed, damaged, reordered
- How do you ensure that you friend receives the letter?



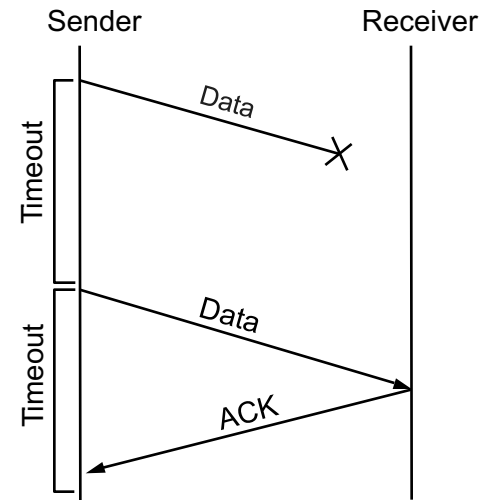
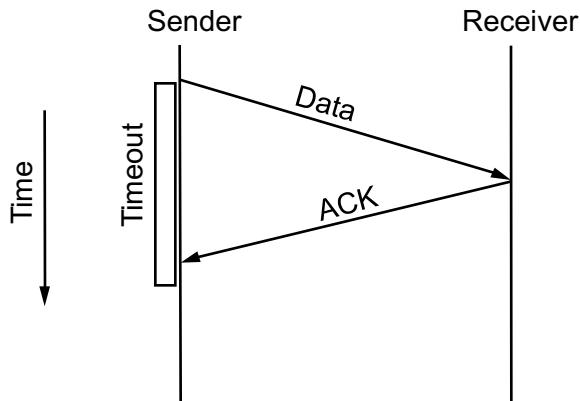


Reliable Transmission

- The data networking version of the same problem
 - ◆ How do we reliably send a message when packets can be lost/corrupted in the network?
- Two options
 - ◆ **Detect a loss/corruption and retransmit**
 - ◆ Send data redundantly to tolerate loss/corruption



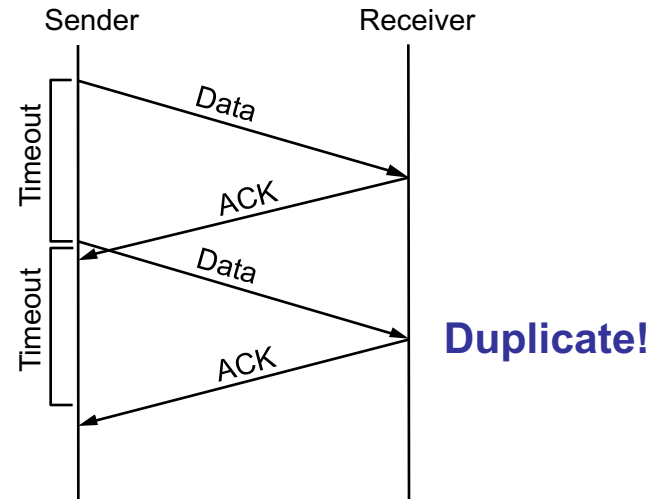
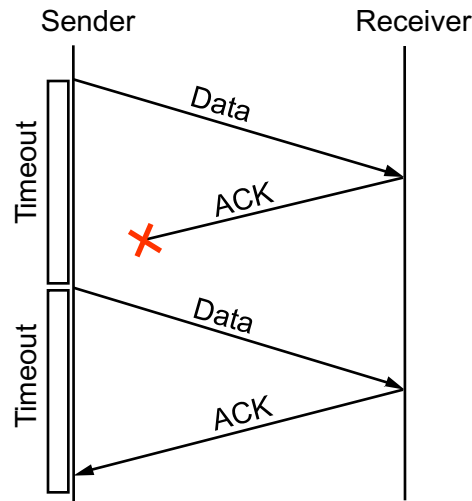
Simple Idea: ARQ



- 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)



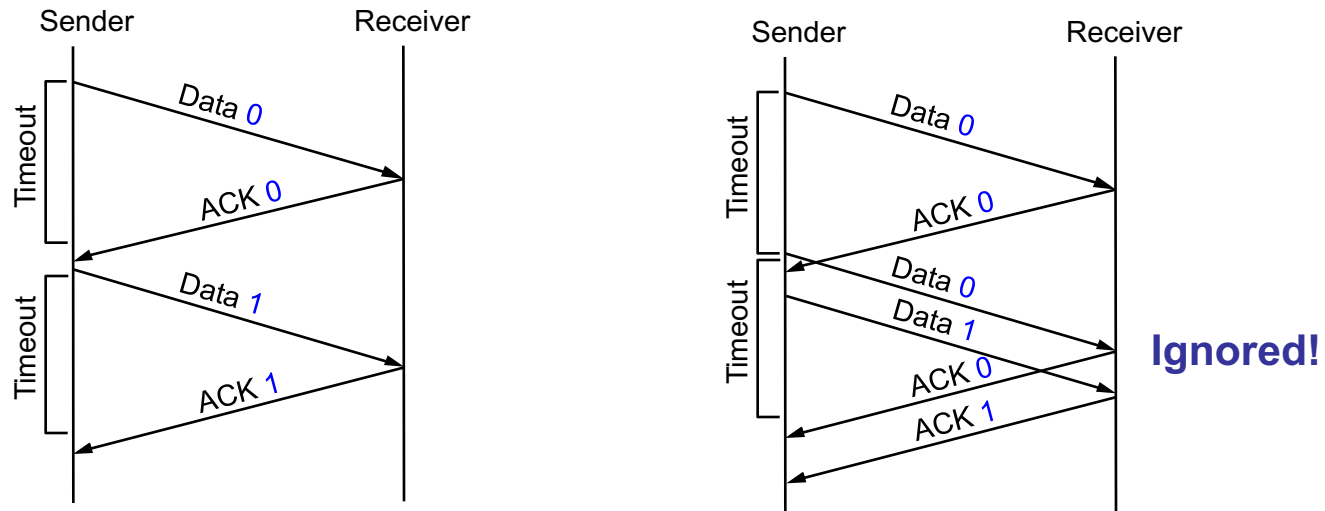
Not So Fast...



- 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

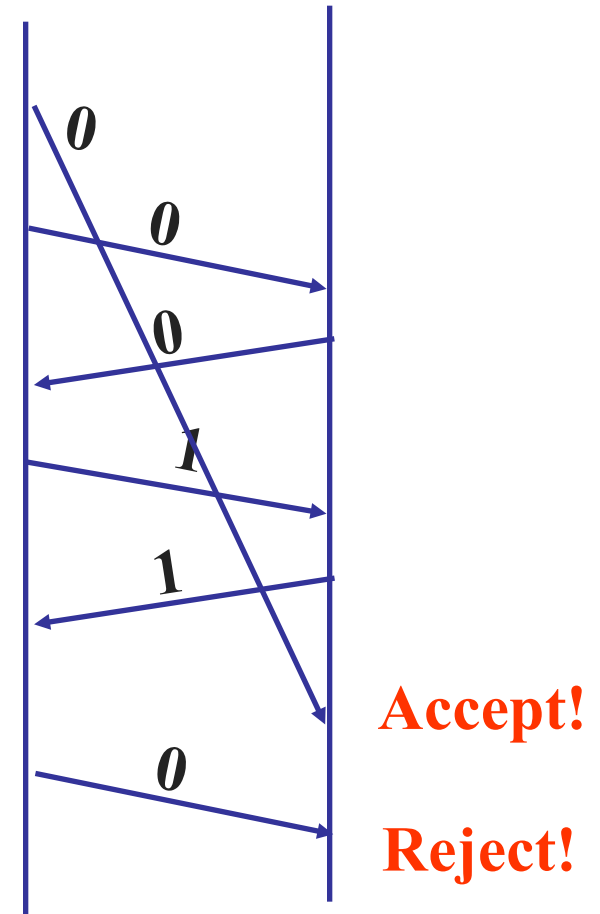


- Sequence numbers solve this problem
 - ◆ Receiver can simply ignore duplicate data
 - ◆ But must still send an ACK! (Why?)
- Simplest ARQ: **Stop-and-wait**
 - ◆ Only one outstanding frame at a time

What if packets are delayed?



- One bit not enough... what to do?
- Never reuse a seq #?
 - ◆ Seq #s could be really big
- Require in-order delivery?
 - ◆ Hard to guarantee in some networks
- Prevent very late delivery?
 - ◆ Limit lifetime of each packet (drop pkt if not delivered in n seconds)
 - ◆ Seq #s not reused within delay bound
 - ◆ Approximate with big seq #s





For Next Time

- Sliding Windows
 - ◆ Read 2.6 in P&D
 - ◆ Guest Lecture: Geoff Voelker
- HW #1 due (at beginning of class)