CSCI-1680
Link Layer Reliability

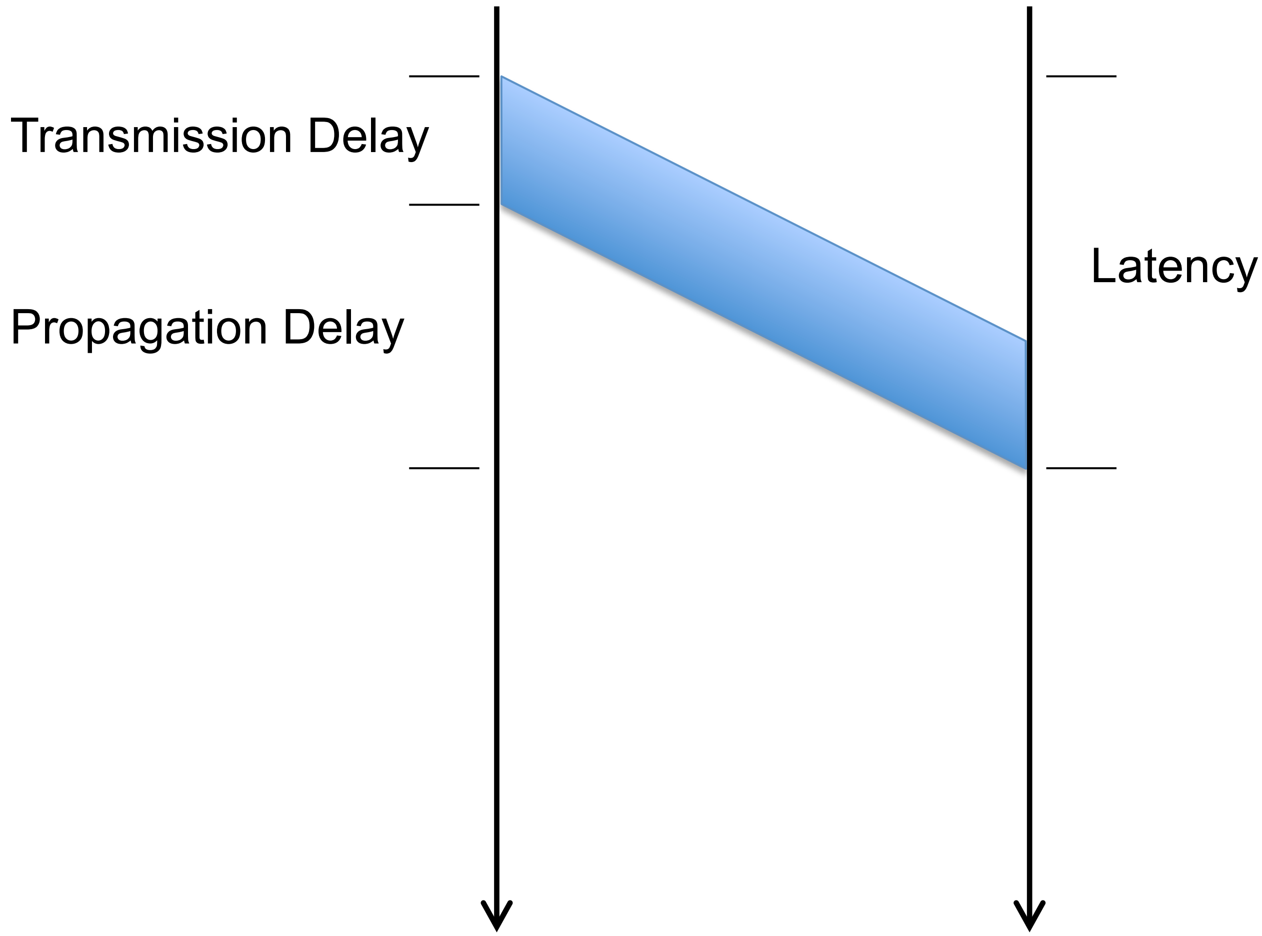
John Jannotti

Based partly on lecture notes by David Mazières, Phil Levis, Rodrigo Fonseca

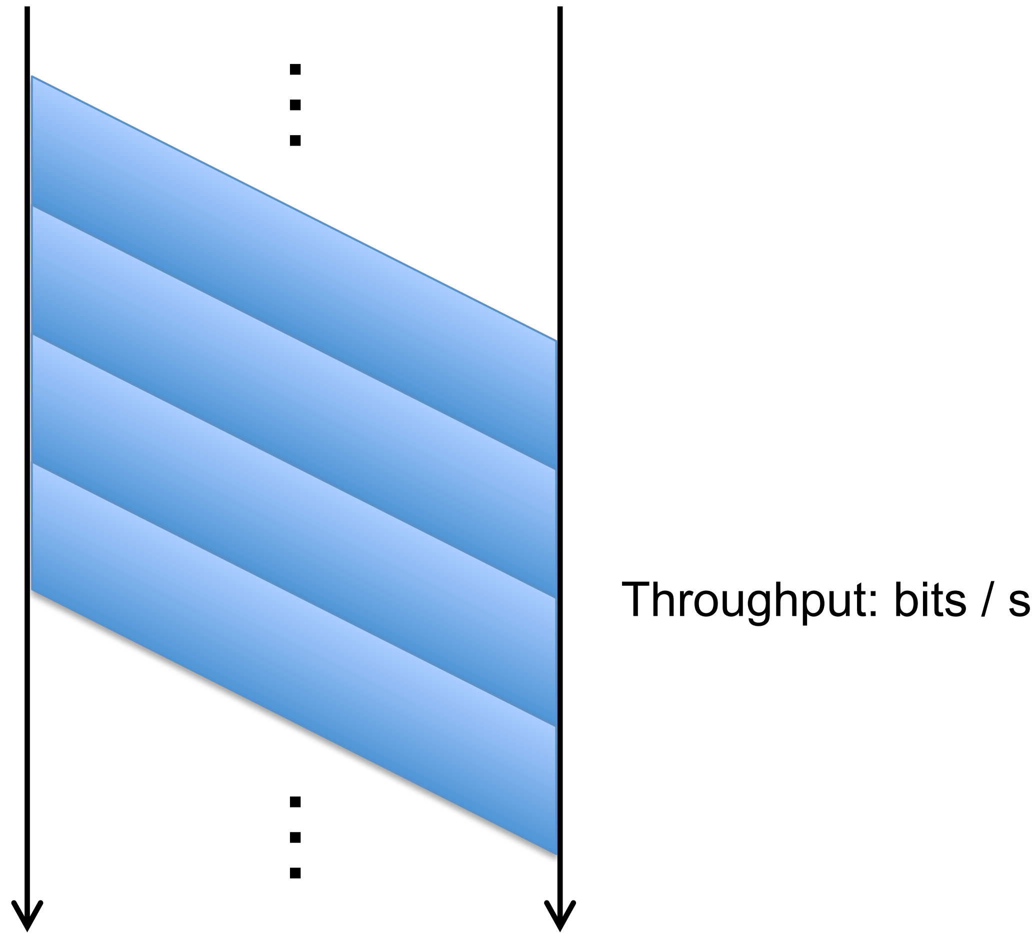
Roadmap

- **Last time**
  - Physical layer: encoding, modulation
  - Link layer framing
- **Today**
  - Getting frames across: reliability, performance

Sending Frames Across

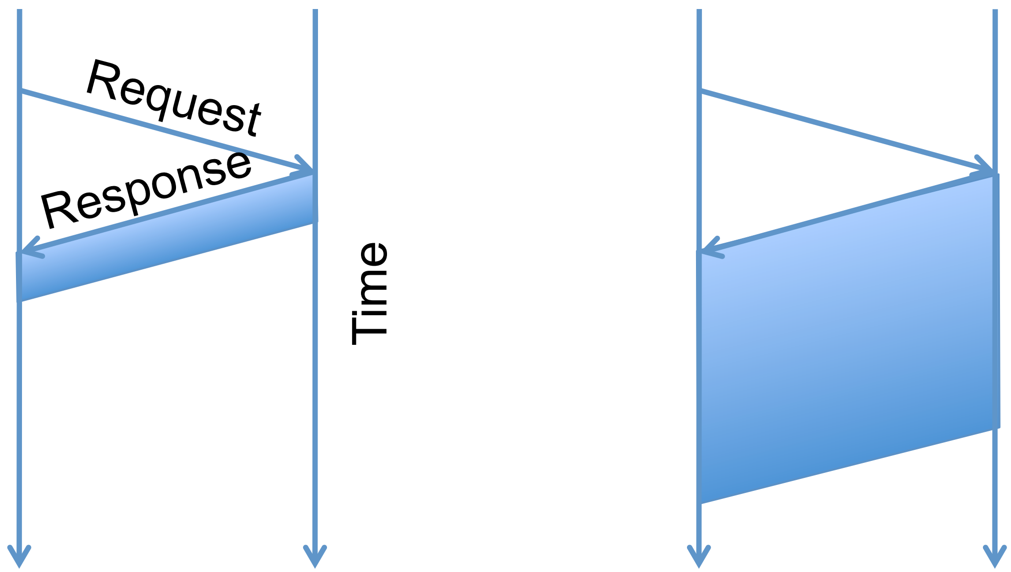


Sending Frames Across



Which matters more, bandwidth or delay?

- How much data can we send during one RTT?
- *E.g.*, send request, receive file



- For small transfers, latency more important, for bulk, throughput more important

Performance Metrics

- **Throughput - Number of bits received/unit of time**
  - *e.g.* 10Mbps
- **Goodput - *Useful* bits received per unit of time**
- **Latency – How long for message to cross network**
  - Process + Queue + Transmit + Propagation
  - Time to First Byte (TTFB) – “goodput’ version of latency
- **Jitter – Variation in latency**
  - Most important for **real-time** media (voice, video conf)

Latency

- **Processing**
  - Per message, small, limits throughput
  - *e.g.* $\frac{100Mb}{s} \times \frac{pkt}{1500B} \times \frac{B}{8b} \approx 8{,}333 pkt/s$ or 120μs/pkt
- **Queue**
  - Highly variable, offered load vs outgoing b/w
- **Transmission**
  - Size/Bandwidth
- **Propagation**
  - Distance/Speed of Light in Medium
  - Cat5 cable: 0.64c

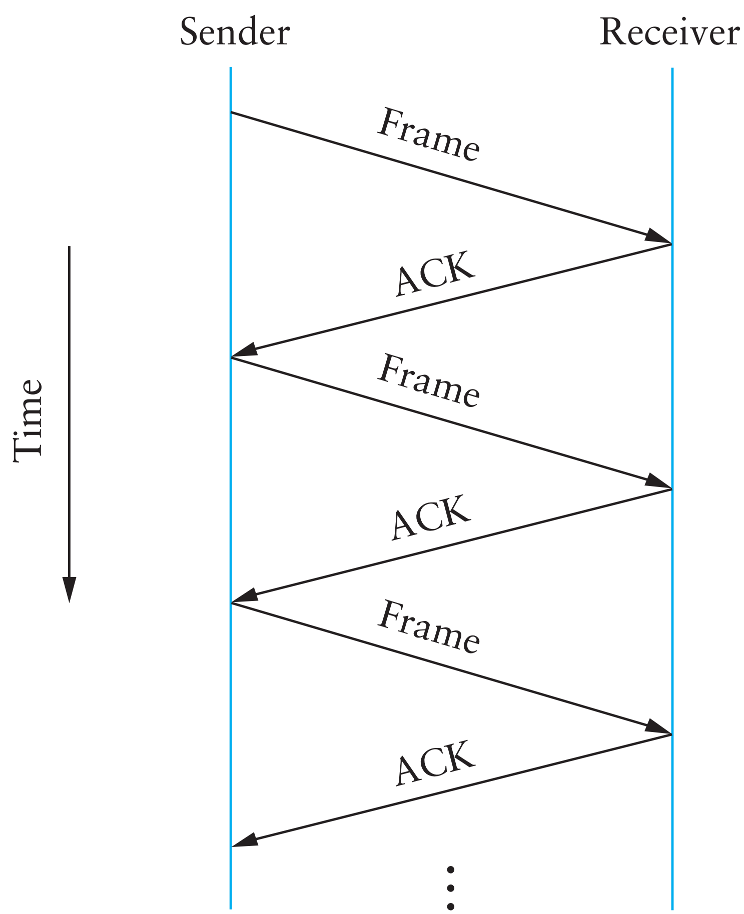
Reliable Delivery

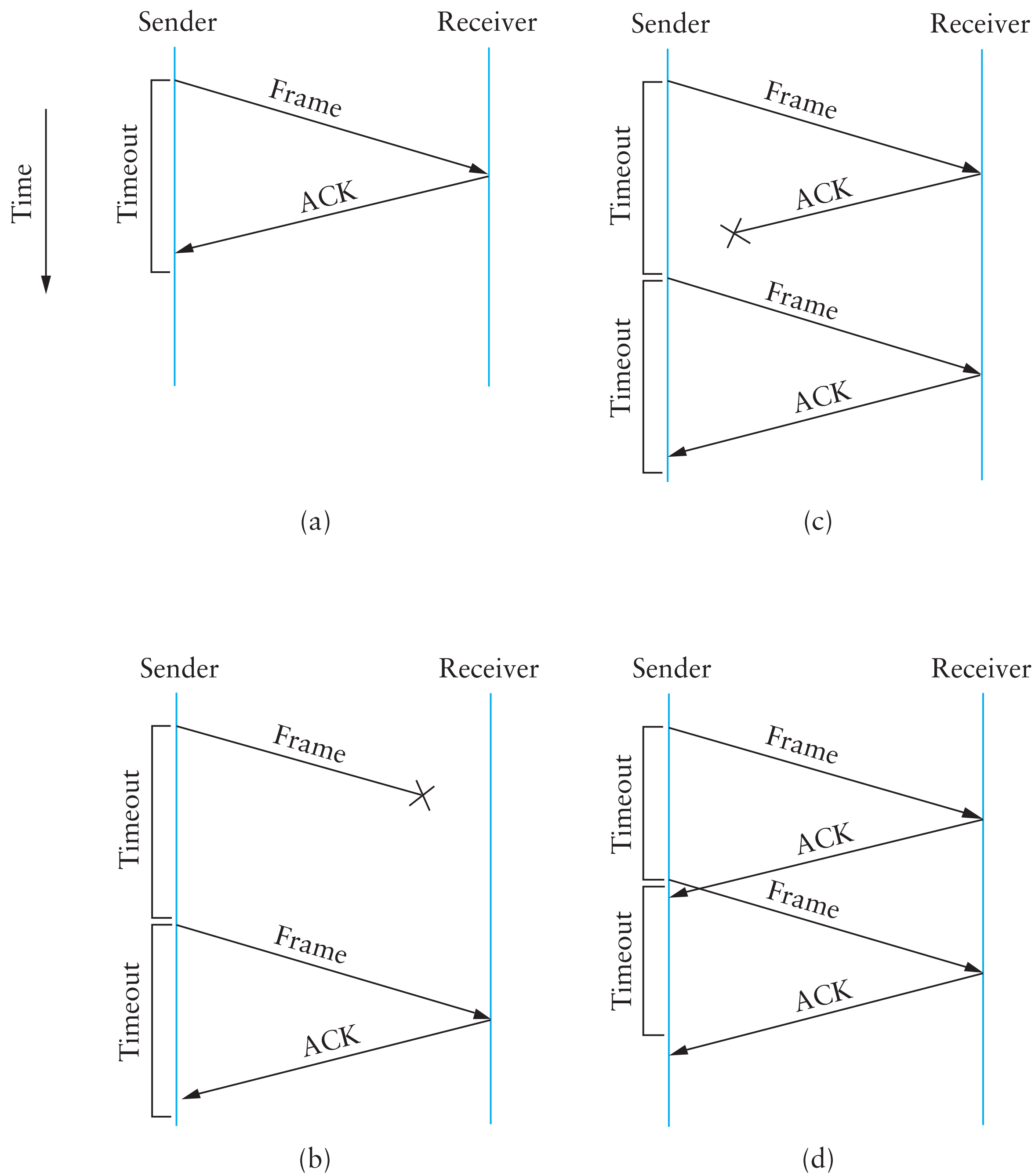
- **Several sources of errors in transmission**
- **Error detection can discard bad frames**
- **Problem: if bad packets are lost, how can we ensure reliable delivery?**
  - Exactly-once semantics = at least once + at most once

At Least Once Semantics

- **How can the sender know packet arrived *at least once*?**
  - Acknowledgments + Timeout
- **Stop and Wait Protocol**
  - S: Send packet, wait
  - R: Receive packet, send ACK
  - S: Receive ACK, send next packet

    or
  - S: Time passes + No ACK, retransmit

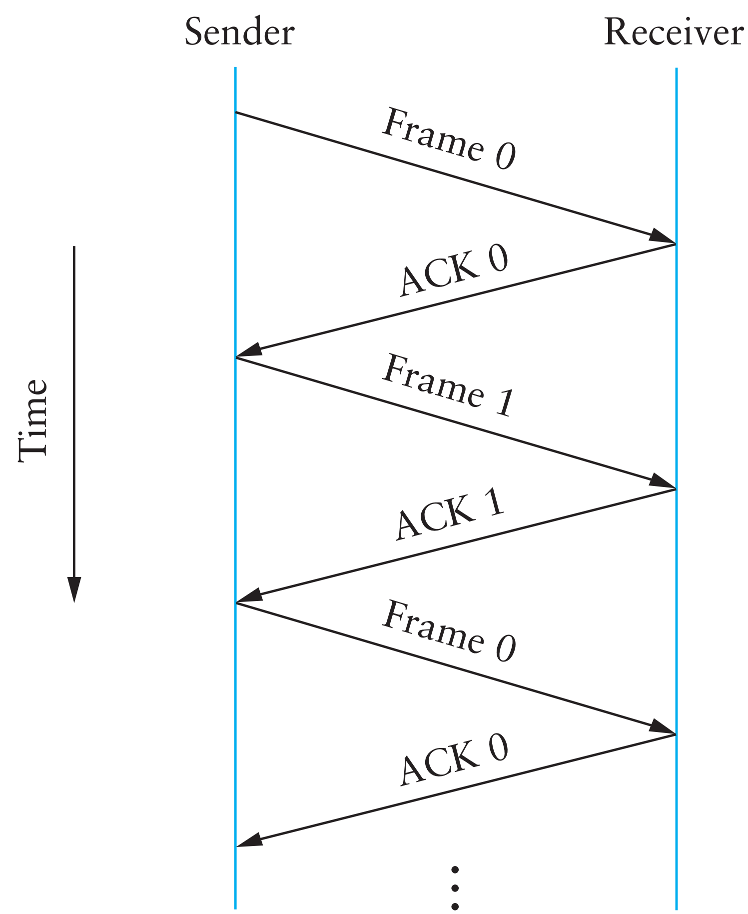

Sender
Receiver
Frame
ACK
Frame
ACK
Frame
ACK
Time


Time
Sender
Receiver
Timeout
Frame
ACK
(a)
Sender
Receiver
Timeout
Frame
ACK
Timeout
Frame
ACK
(c)
Sender
Receiver
Timeout
Frame
Timeout
Frame
ACK
(b)
Sender
Receiver
Timeout
Frame
ACK
Timeout
Frame
ACK
(d)

Stop and Wait Problems

- Duplicate data
- Duplicate acks
- Slow (channel idle most of the time!)
- May be difficult to set the timeout value

Duplicate data: adding sequence numbers

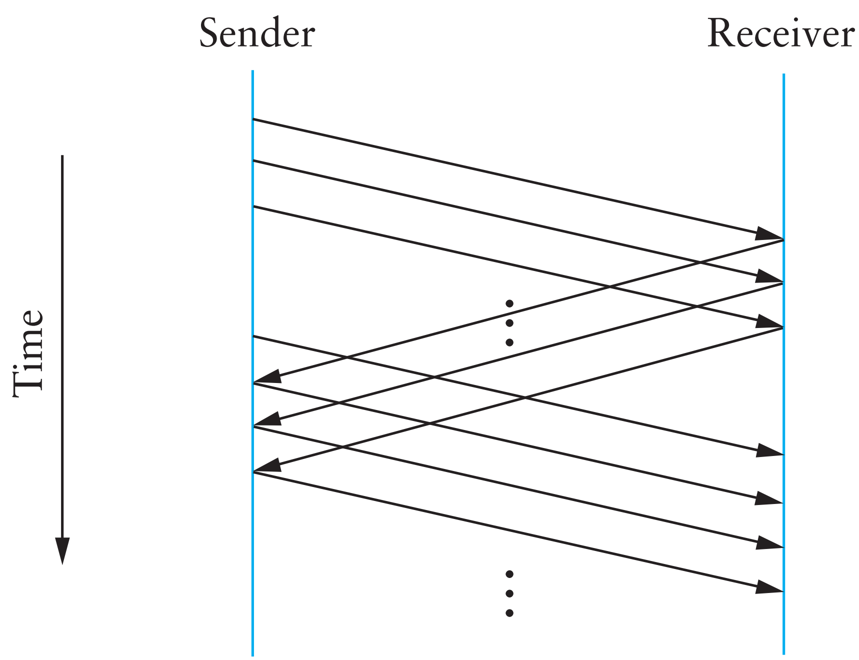


At Most Once Semantics

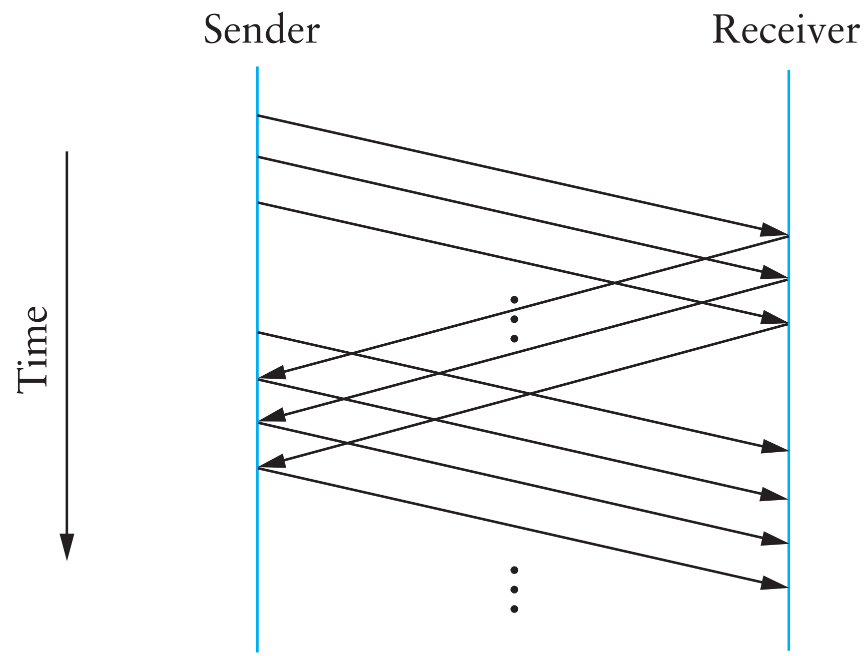
- **How to avoid duplicates?**
  - Uniquely identify each packet
  - Have receiver and sender remember
- **Stop and Wait: add 1 bit to the header**
  - Why is it enough?

Going faster: sliding window protocol

- **Still have the problem of keeping pipe full**
  - Generalize approach with > 1-bit counter
  - Allow multiple outstanding (unACKed) frames
  - Upper bound on unACKed frames, called *window*

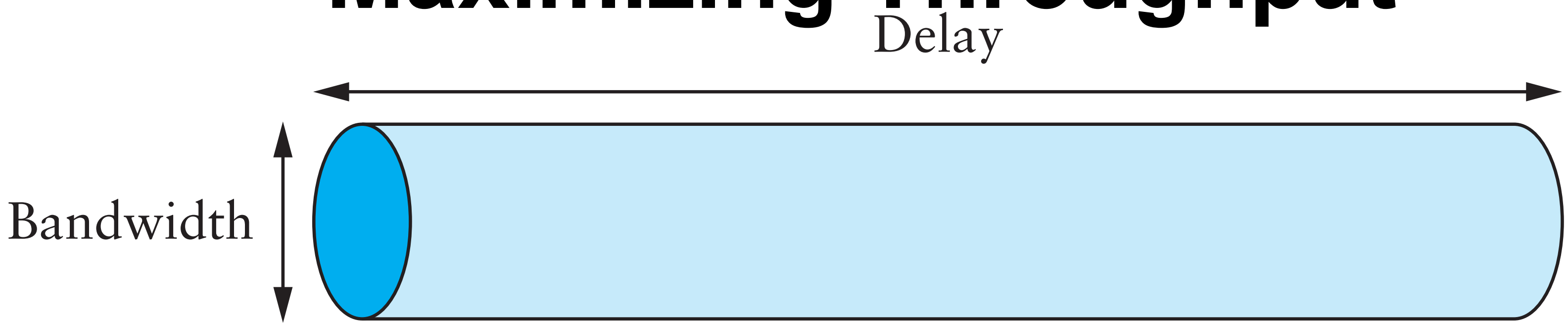


How big should the window be?



- **How many bytes can we transmit in one RTT?**
  – BW B/s x RTT s => "Bandwidth-Delay Product"

Maximizing Throughput

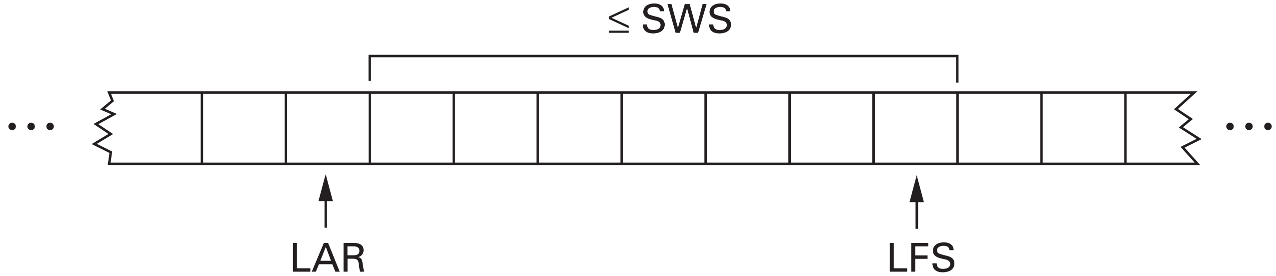


- **Can view network as a pipe**
  - For full utilization want:

    potential bytes in flight ≥ bandwidth × delay
  - But don’t want to overload the network (congestion control - future lectures)
- **What if protocol doesn’t involve bulk transfer?**
  - Get throughput through concurrency – service multiple clients simultaneously

Sliding Window Sender

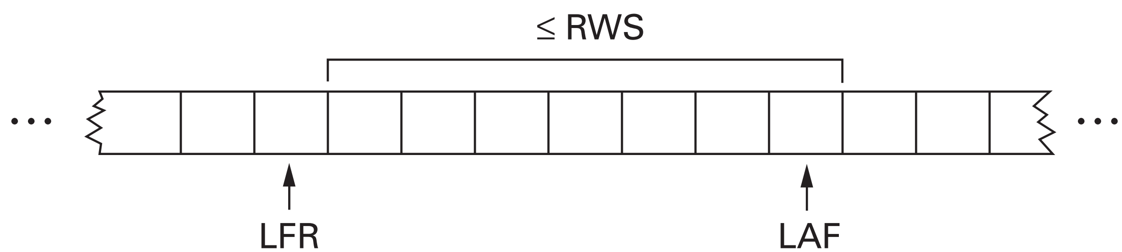
- **Assign sequence number (SeqNum) to each frame**
- **Maintain three state variables**
  - send window size (SWS)
  - last acknowledgment received (LAR)
  - last frame sent (LFS)



- **Maintain invariant: LFS – LAR ≤ SWS**
- **Advance LAR when ACK arrives**
- **Buffer up to SWS frames**

Sliding Window Receiver

- **Maintain three state variables:**
  - receive window size (RWS)
  - largest acceptable frame (LAF)
  - last frame received (LFR)



- **Maintain invariant: LAF – LFR ≤ RWS**
- **Frame SeqNum arrives:**
  - if LFR < SeqNum ≤ LAF, accept
  - if SeqNum ≤ LFR or SeqNum > LAF, discard
- **In simplest version: Send *cumulative* ACKs**

Tuning Send Window

- **How big should SWS be?**
  - "Fill the pipe" (bw x delay)
- **How big should RWS be?**
  - $1 \leq RWS \leq SWS$
- **How many distinct sequence numbers needed?**
  - SWS can't be more than *half* of the space of valid seq#s. Why?

Example

- SWS = RWS = 5. Are 6 seq #s enough?
- Sender sends 0,1,2,3,4
- All acks are lost
- Sender sends 0,1,2,3,4 again
- …

Summary

- **Want exactly once**
  - At least once: acks + timeouts + retransmissions
  - At most once: sequence numbers
- **Want efficiency**
  - Sliding window

Error Detection

- **Idea: have some codes be *invalid***
  - Add bits to catch errors in packet (802.15.4)
- **Sometimes can also *correct* errors**
  - If enough redundancy (chips “almost” match)
  - Otherwise, count on retransmission
- **Used in multiple layers**
- **Three examples today:**
  - Parity
  - Internet Checksum
  - CRC

Simplest Schemes

- **Repeat frame *n* times**
  - Can we detect errors?
  - Can we correct errors?
    - Voting
  - Problem: high redundancy : *n*
- **Example: send each bit 3 times**
  - Valid codes: 000 111
  - Invalid codes : 001 010 011 100 101 110
  - Corrections : 0 0 1 0 1 1

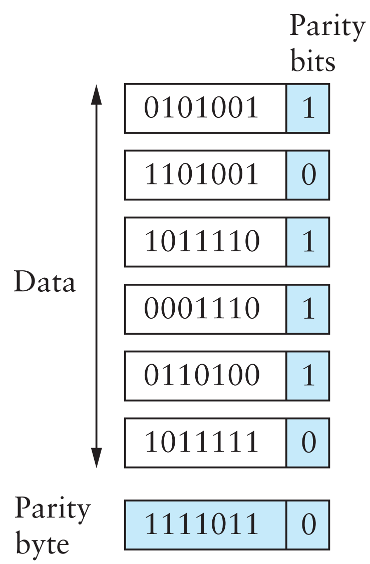
Parity

- **Add a parity bit to the end of a word**
- **Example with 2 bits:**
  - Valid: 000 011 011 110
  - Invalid: 001 010 010 111
  - Can we correct?
- **Can detect odd number of bit errors**
  - No correction

In general

- **Hamming distance: number of bits that are different**
  – E.g.: HD (00001010, 01000110) = 3
- **If min HD between valid codewords is *d*:**
  – Can detect *d*-1 bit error
  – Can correct $\lfloor (d-1)/2 \rfloor$ bit errors
- **What is *d* for parity and 3-voting?**
- **How can we cleverly select codewords?**

2-D Parity



- **Add 1 parity bit for each 7 bits**
- **Add 1 parity bit for each bit position across the frame)**
  - Can correct single-bit errors
  - Can detect 2- and 3-bit errors, most 4-bit errors
- **Find a 4-bit error that can't be detected**

IP Checksum

- **Fixed-length code**
  - n-bit code should capture all but 2^{-n} fraction of errors
    - Why?
  - Trick is to make sure that includes all *common* errors
- **IP Checksum is an example**
  - 1's complement of 1's complement sum of every 2 bytes

```
uint16 cksum(uint16 *buf, int count) {
    uint32 sum = 0;
    while (count--)
        if ((sum += *buf++) & 0xffff0000) // carry
            sum = (sum & 0xffff) + 1;
    return ~(sum & 0xffff);
}
```

How good is it?

- **16 bits not very long: misses how many errors?**
  - 1 in 2^{16}, or 1 in 64K errors
- **Checksum does catch all 1-bit errors**
- **But not all 2-bit errors**
  - E.g., increment word ending in 0, decrement one ending in 1
- **Checksum also optional in UDP**
  - All 0s means no checksums calculated
  - If checksum word gets wiped to 0 as part of error, bad news

From rfc791 (IP)

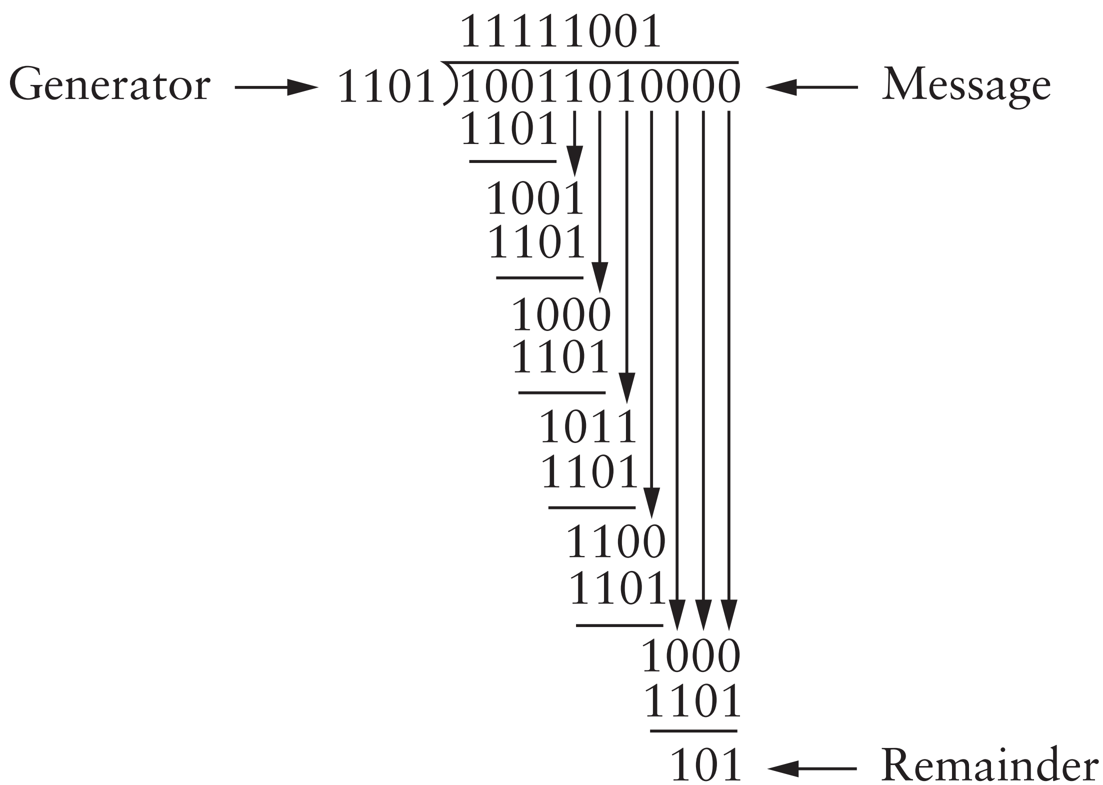
"This is a simple to compute checksum and experimental evidence indicates it is adequate, but it is provisional and may be replaced by a CRC procedure, depending on further experience."

CRC – Error Detection with Polynomials

- **Goal: maximize protection, minimize bits**
- **Consider message to be a polynomial in $Z_2[x]$**
  - Each bit is one coefficient
  - E.g., message 10101001 -> $m(x) = x^7 + x^5 + x^3 + 1$
- **Can reduce one polynomial modulo another**
  - Let $n(x) = m(x)x^3$. Let $C(x) = x^3 + x^2 + 1$.
  - $n(x)$ "**mod**" $C(x)$: $r(x)$
  - Find $q(x)$ and $r(x)$ s.t. $n(x) = q(x)C(x) + r(x)$ and degree of $r(x)$ < degree of $C(x)$
  - Analogous to taking 11 **mod** 5 = 1

Polynomial Division Example

- **Just long division, but addition/subtraction is XOR**



CRC

- **Select a divisor polynomial C(x), degree k**
  - C(x) should be *irreducible* – not expressible as a product of two lower-degree polynomials in $Z_2[x]$
- **Add k bits to message**
  - Let $n(x) = m(x)x^k$ (add *k* 0's to *m*)
  - Compute $r(x) = n(x) \bmod C(x)$
  - Compute $n(x)$ = n(x) – r(x) (will be divisible by C(x)) (subtraction is XOR, just set k lowest bits to r(x)!)
- **Checking CRC is easy**
  - Reduce message by C(x), make sure remainder is 0

Why is this good?

- **Suppose you send m(x), recipient gets m'(x)**
  - E(x) = m'(x) – m(x) (all the incorrect bits)
  - If CRC passes, C(x) divides m'(x)
  - Therefore, C(x) must divide E(x)
- **Choose C(x) that doesn't divide any common errors!**
  - All single-bit errors caught if x^k, x^0 coefficients in C(x) are 1
  - All 2-bit errors caught if at least 3 terms in C(x)
  - Any odd number of errors if last two terms (x + 1)
  - Any error burst less than length k caught

Common CRC Polynomials

- **Polynomials not trivial to find**
  - Some studies used (almost) exhaustive search
- **CRC-8: $x^8 + x^2 + x^1 + 1$**
- **CRC-16: $x^{16} + x^{15} + x^2 + 1$**
- **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 easily computable in hardware**

An alternative for reliability

- **Erasure coding**
  - Assume you can detect errors
  - Code is designed to tolerate entire missing frames
    - Collisions, noise, drops because of bit errors
  - Forward error correction
- **Examples: Reed-Solomon, Tornado, LT, & Raptor Codes**
- **Property:**
  - K source frames, produce B > K encoded frames
  - Receiver can reconstruct source with *any* K' frames, with K' *slightly* larger than K
  - Some codes can make B as large as needed.
  - Amazingly, this sorcery isn't that hard to understand!

Next class

- **Link Layer II**
  - Ethernet: dominant link layer technology
    - Framing, MAC, Addressing
  - Switching