COMP/ELEC 429/556 Introduction to Computer Networks

Reliability

Some slides used with permissions from Edward W. Knightly, T. S. Eugene Ng, Ion Stoica, Hui Zhang


Horton
Hatches
the
Egg
Dr. Seuss
Oceanhouse Media
Dr. Seuss
"An elephant's faithful one hundred percent"

Cannot assume network is reliable

- Goal: Realize the reliable byte stream model
  - Fact: most data networks transmit in granularity of packet – a formatted unit of data

- New goal: Reliably transmit packets and maintain packet order

Assumptions:

- Packets can get lost or reordered in the network
- Bits in packets can get corrupted on the way

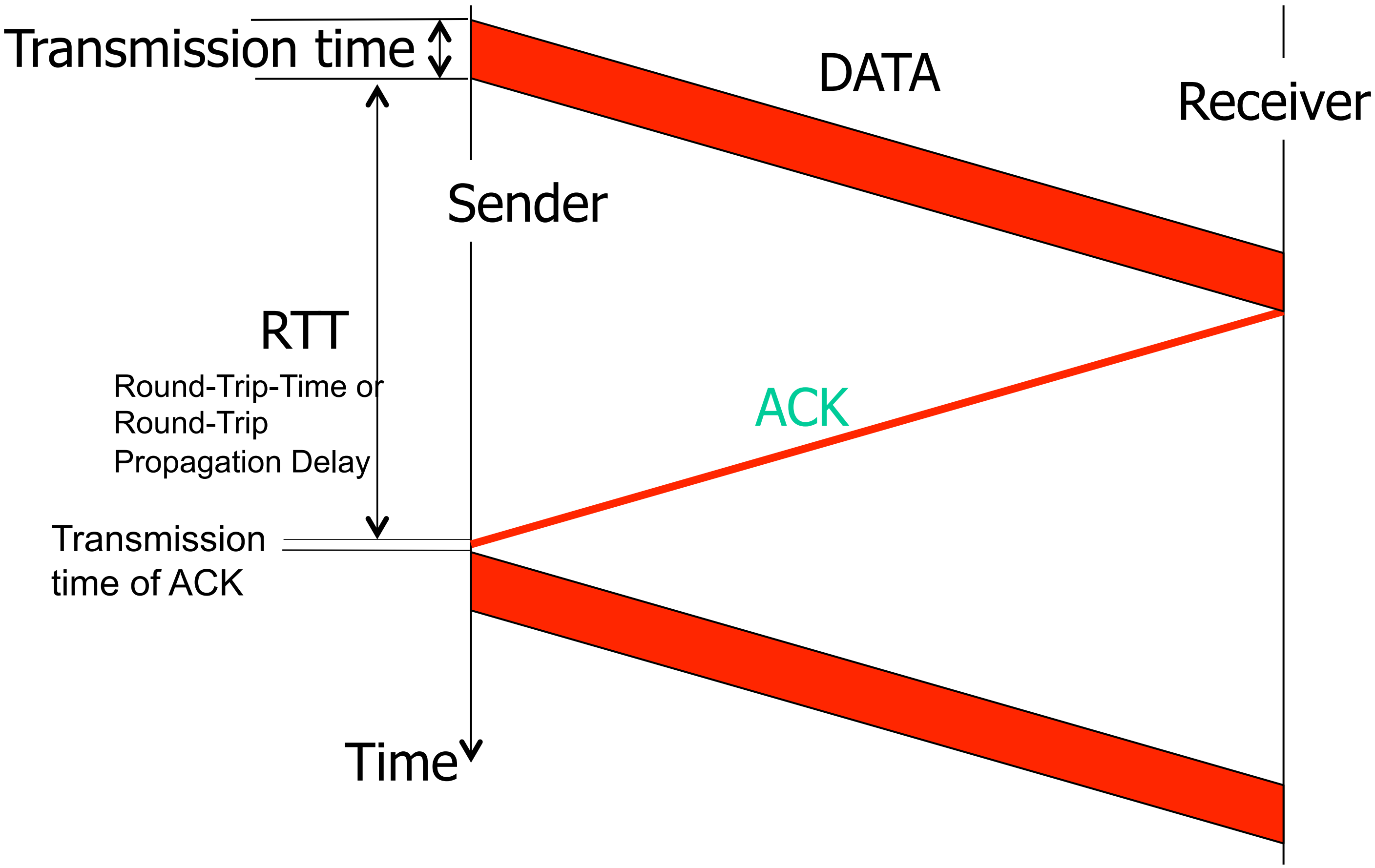
Solutions?

Retransmission-Based Recovery

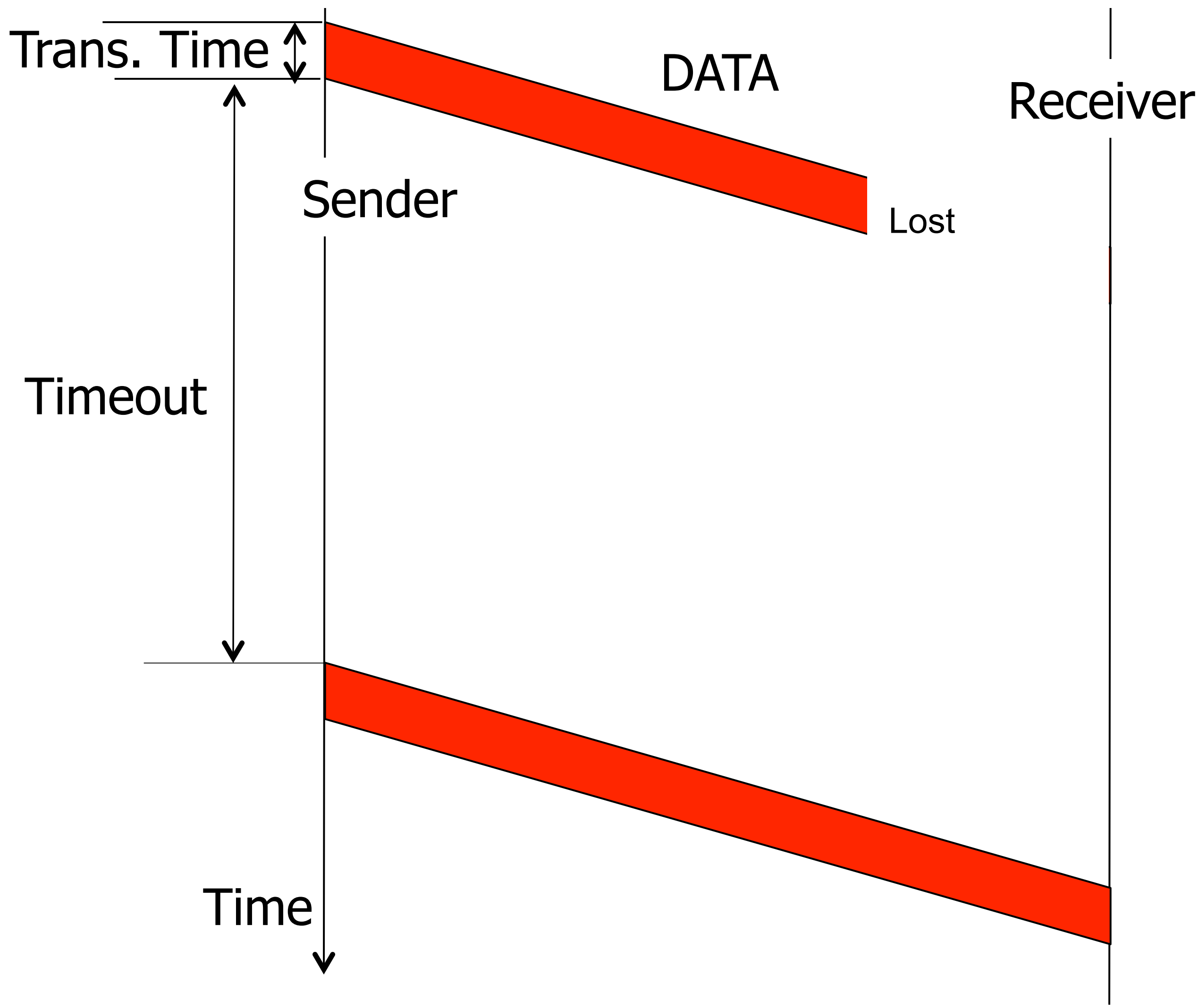
- Retransmit lost or corrupted packets
  - Packets are uniquely identified by sequence numbers
  - Correctly received packets are acknowledged
  - Packets not acknowledged are retransmitted
- Key: Do this efficiently
  - Use available bandwidth efficiently
  - Detect loss/corruption and retransmit quickly

Naïve Approach: Stop-and-Wait

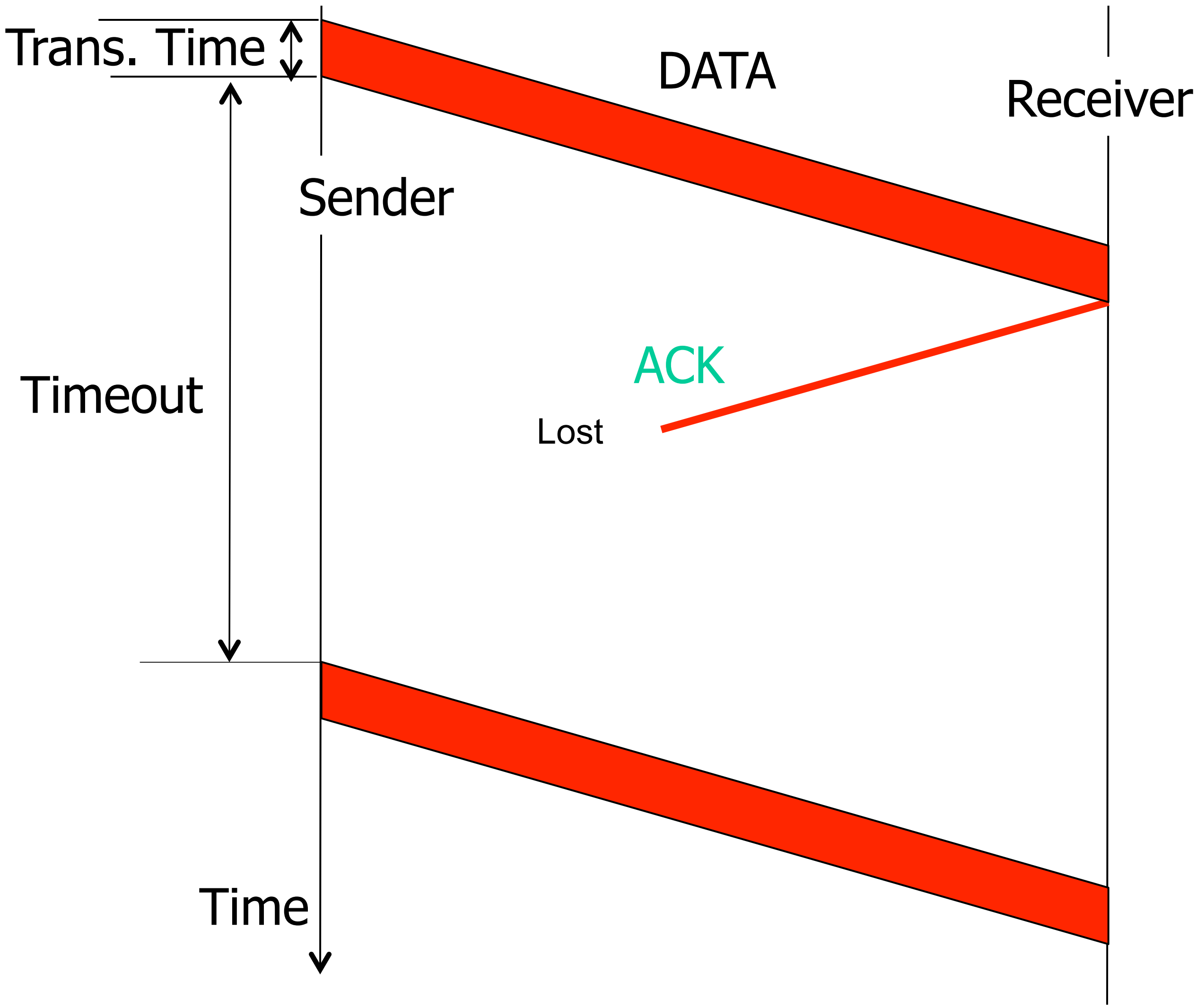
- Send DATA packet; wait for acknowledgement (ACK); repeat
- If timeout, retransmit DATA packet



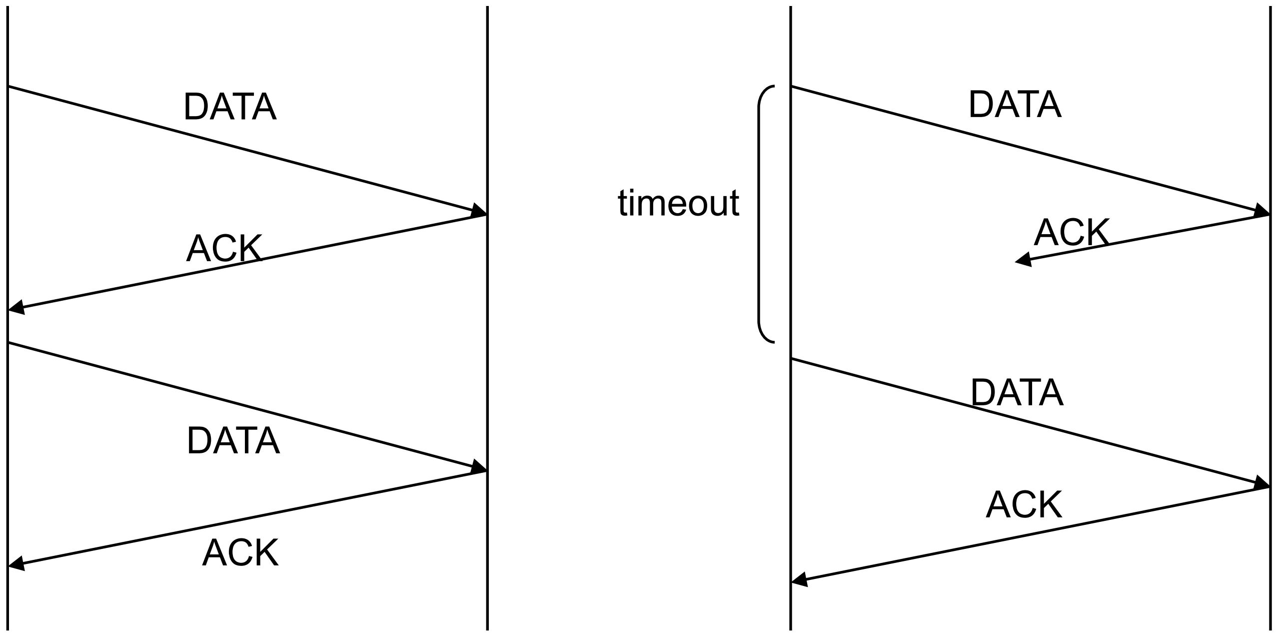
DATA Loss



ACK Loss



How Many Sequence Numbers Needed for Stop-and-Wait?



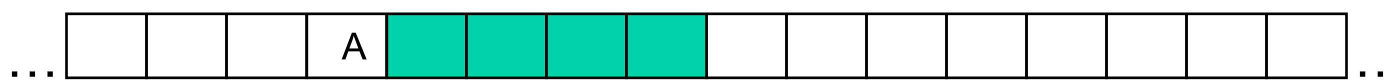
- 2; need a 1 bit sequence number (i.e. alternate between 0 and 1) to distinguish duplicate packets

Problem with Stop-and-Wait

- Lots of time wasted in waiting for acknowledgements
- What if you have a 10Gbps link, a data packet size of 1500B (like Ethernet), ACK size of 40B and a propagation delay of 10ms?
- Because you send one packet and wait
  - Throughput = 1500*8bit/(2*10ms + (1500B+40B)/10Gbps) ~= 600Kbps!
  - A utilization of 0.006%

Better Approach: Sliding Window

- *window* = set of adjacent sequence numbers
- The size of the set is the *window size (denote it n)*
- Let A be the last ack'd packet seq # of sender without gap; then window of sender = {A+1, A+2, …, A+n}

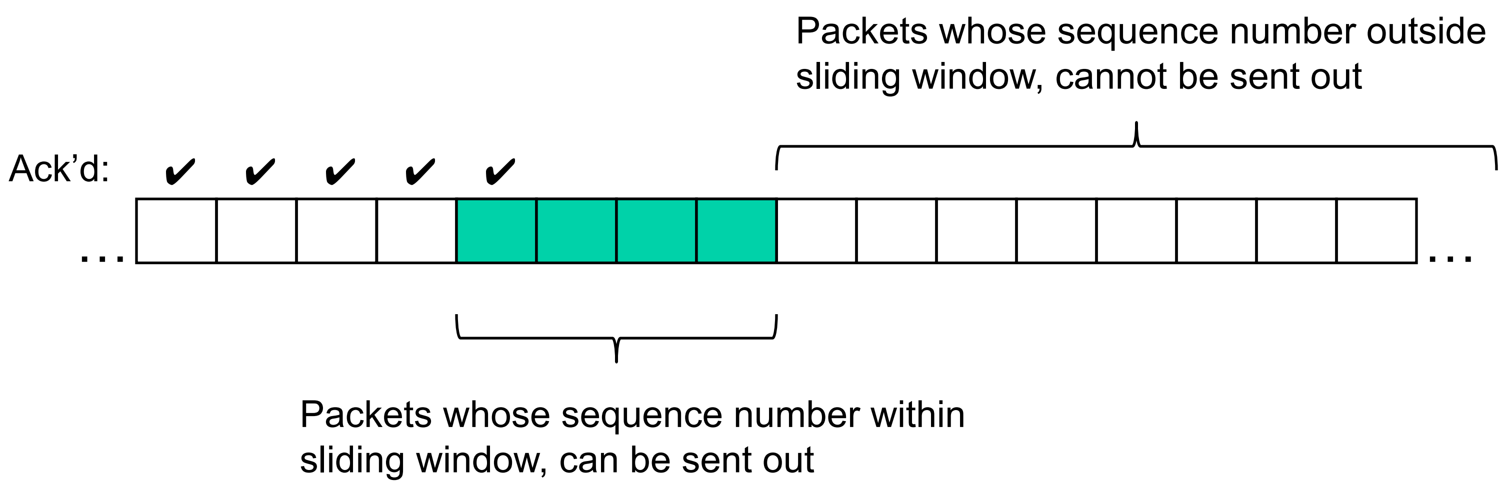


- Sender can send a packet as soon as it is inside its window

- Let B be the last received packet seq # without gap by receiver; then window of receiver = {B+1,…, B+n}

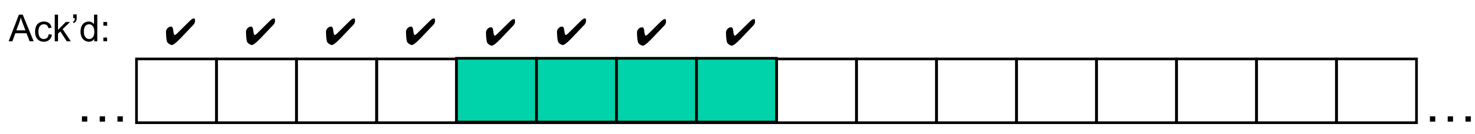
- Receiver can accept out of sequence packets if in window

Sender Sliding Window Example

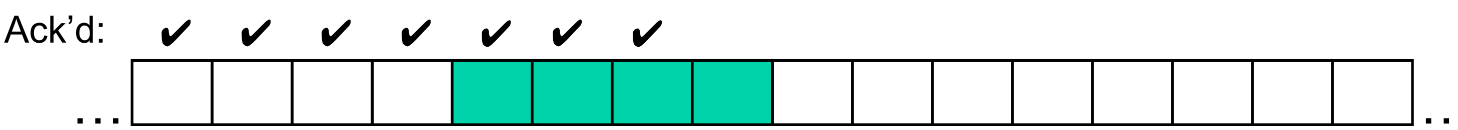


If receive ACK for packet that falls outside of sliding window, ignore such ACK

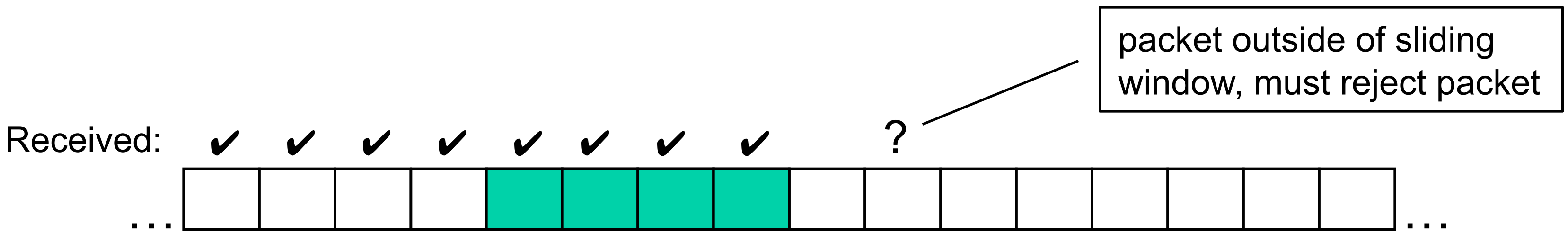
Sender Sliding Window Example



Sender Sliding Window Example

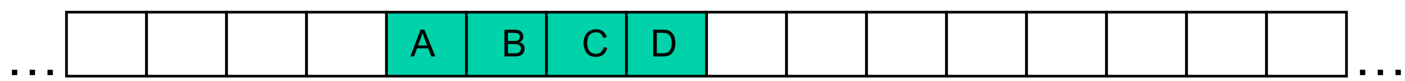


Receiver Sliding Window Example



Basic Timeout and Acknowledgement

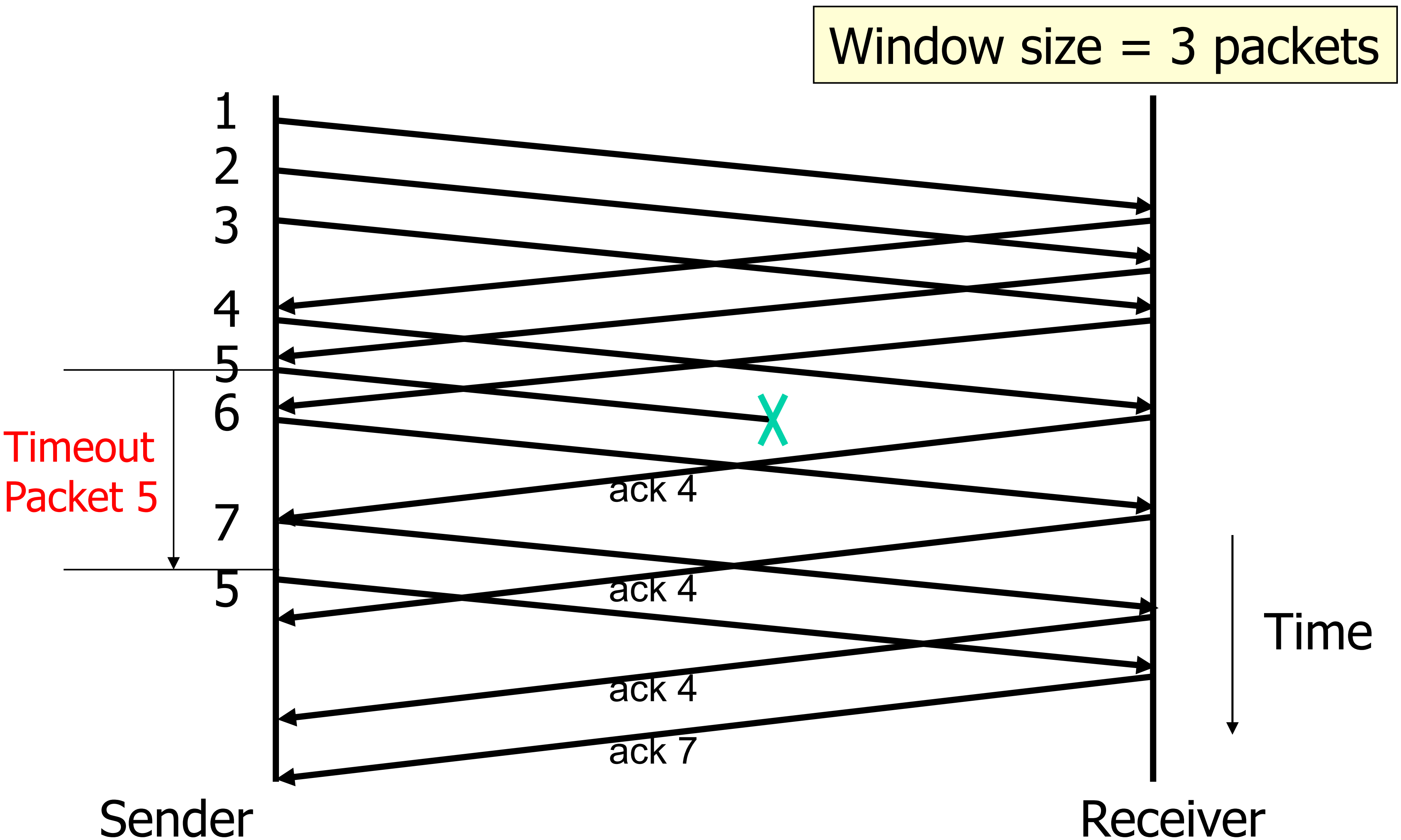
- Every packet k transmitted is associated with a timeout, denoted timeout(k)
- Basic acknowledgement scheme
  – Upon receiving a packet from sender, send an ack for k, where k is the newest packet seq # such that all packets with seq numbers older than or equal to k have been accepted



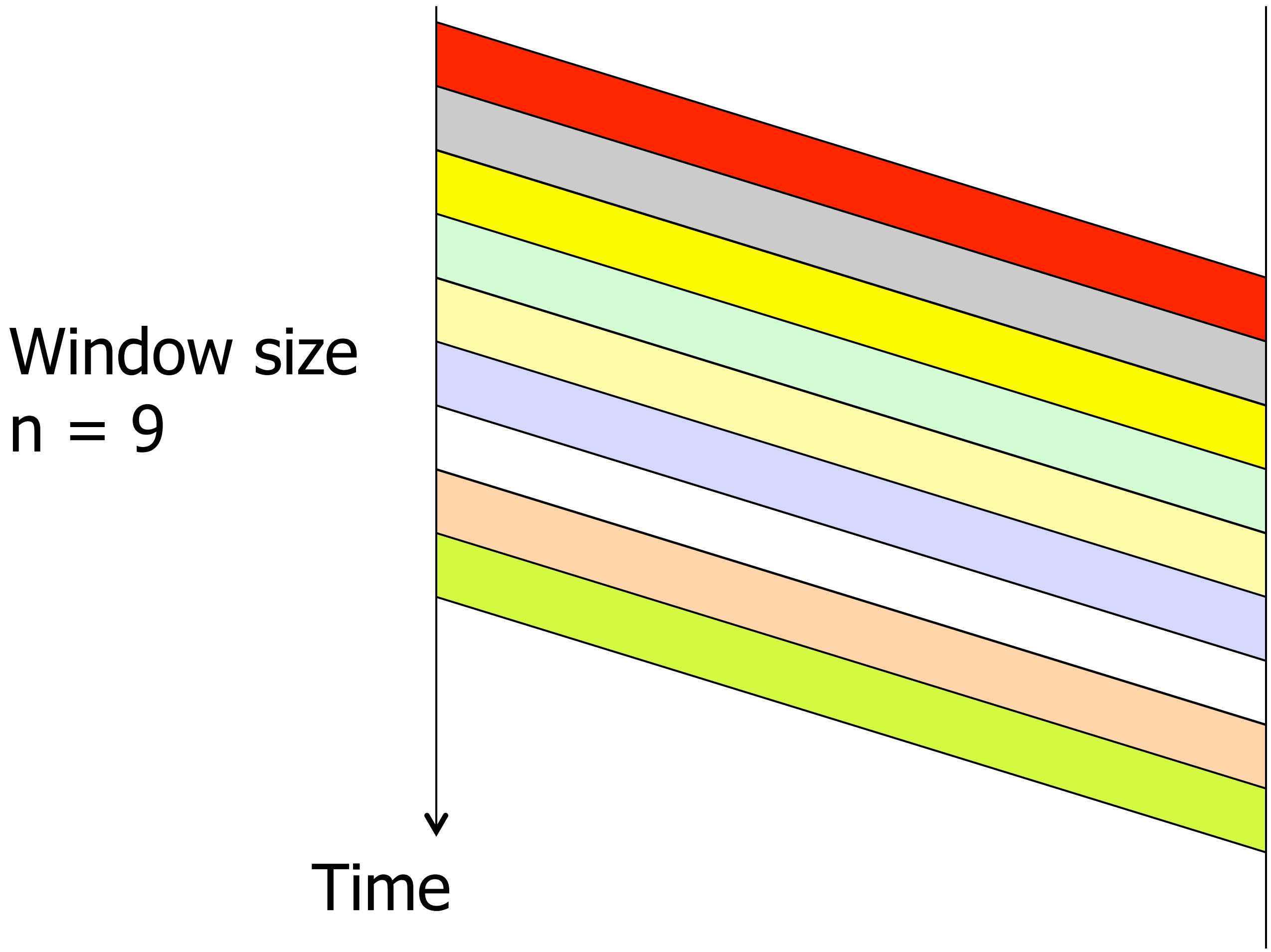
  – Suppose packet B is received but A is missing, ack for A-1 is sent. If C is received next, ack for A-1 is sent again. If D is received next, ack for A-1 is sent again. If A is received next, an ack for D is sent, and the receiver window slides

- If by timeout(k), ack for k or larger has not yet been received, the sender retransmits packet k

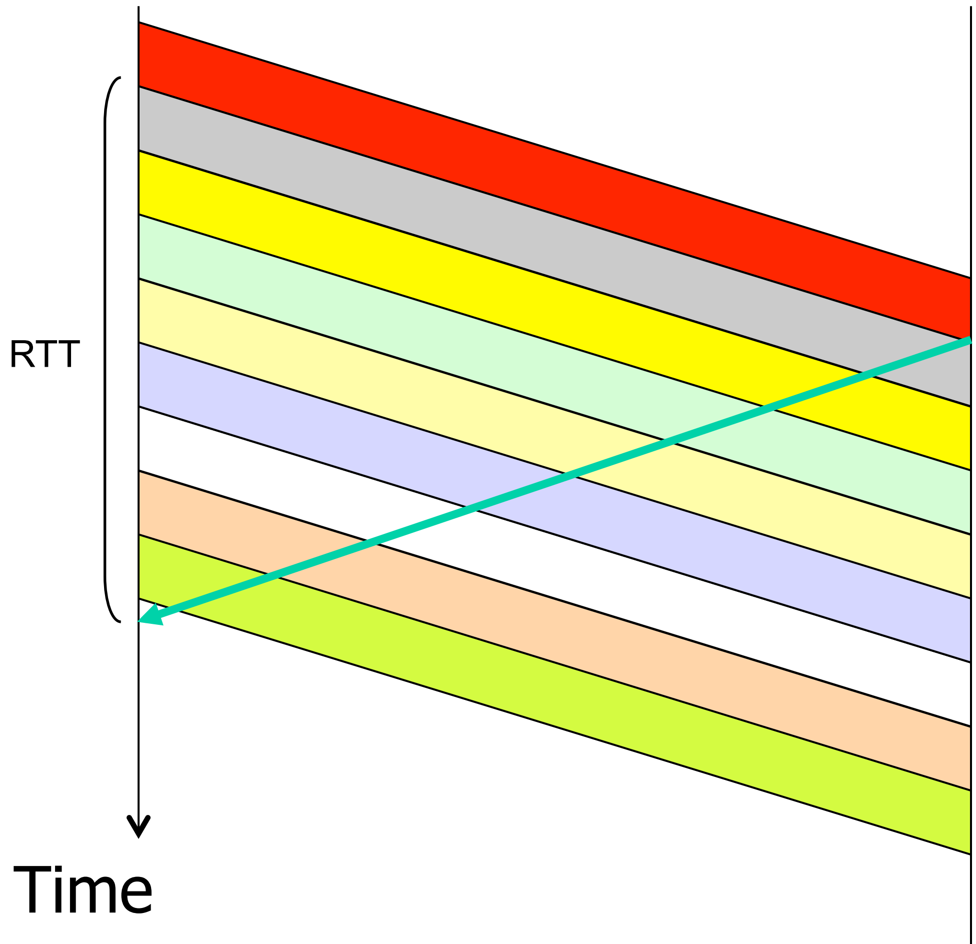
Example with Loss



Packets within Sender Window Can Be Sent



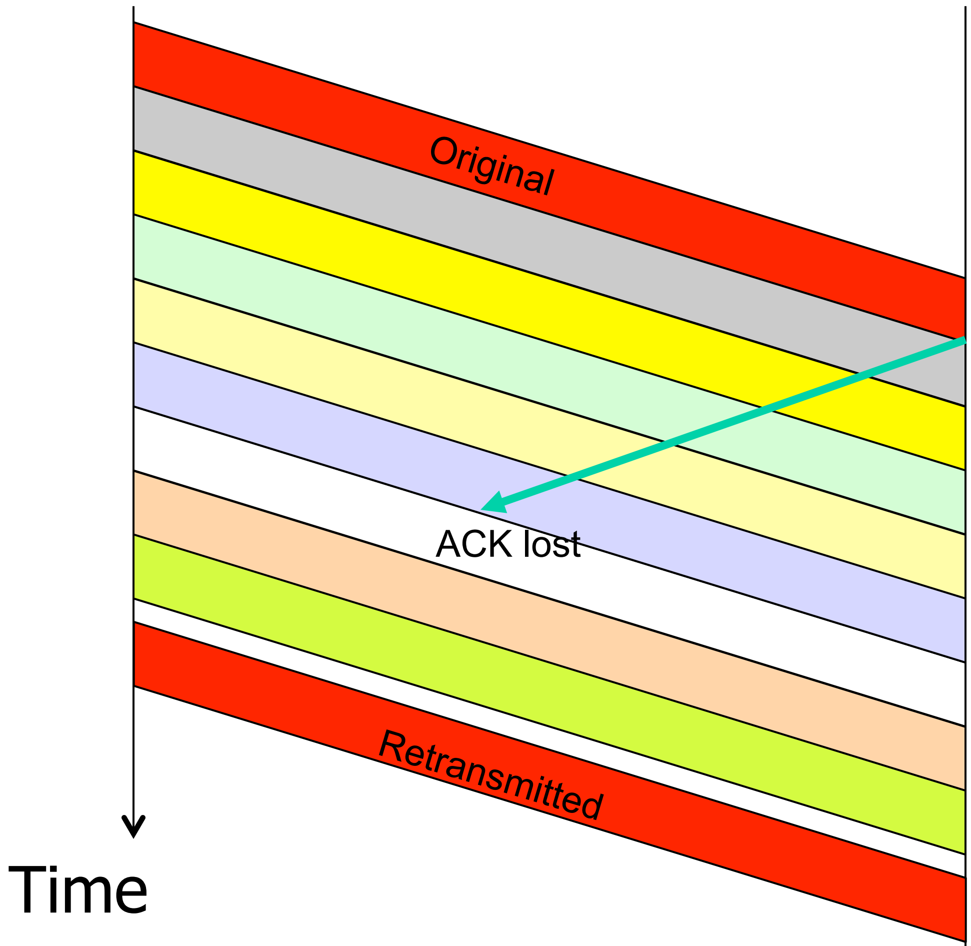
Efficiency



n = 9, i.e. 9 packets in one RTT

Can fully use available bandwidth if n is large enough

How Many Unique Sequence Numbers Needed?



n unique sequence numbers not enough to differentiate the retransmission of a packet in current window and a new transmission from the next window

Need 2*n unique sequence numbers

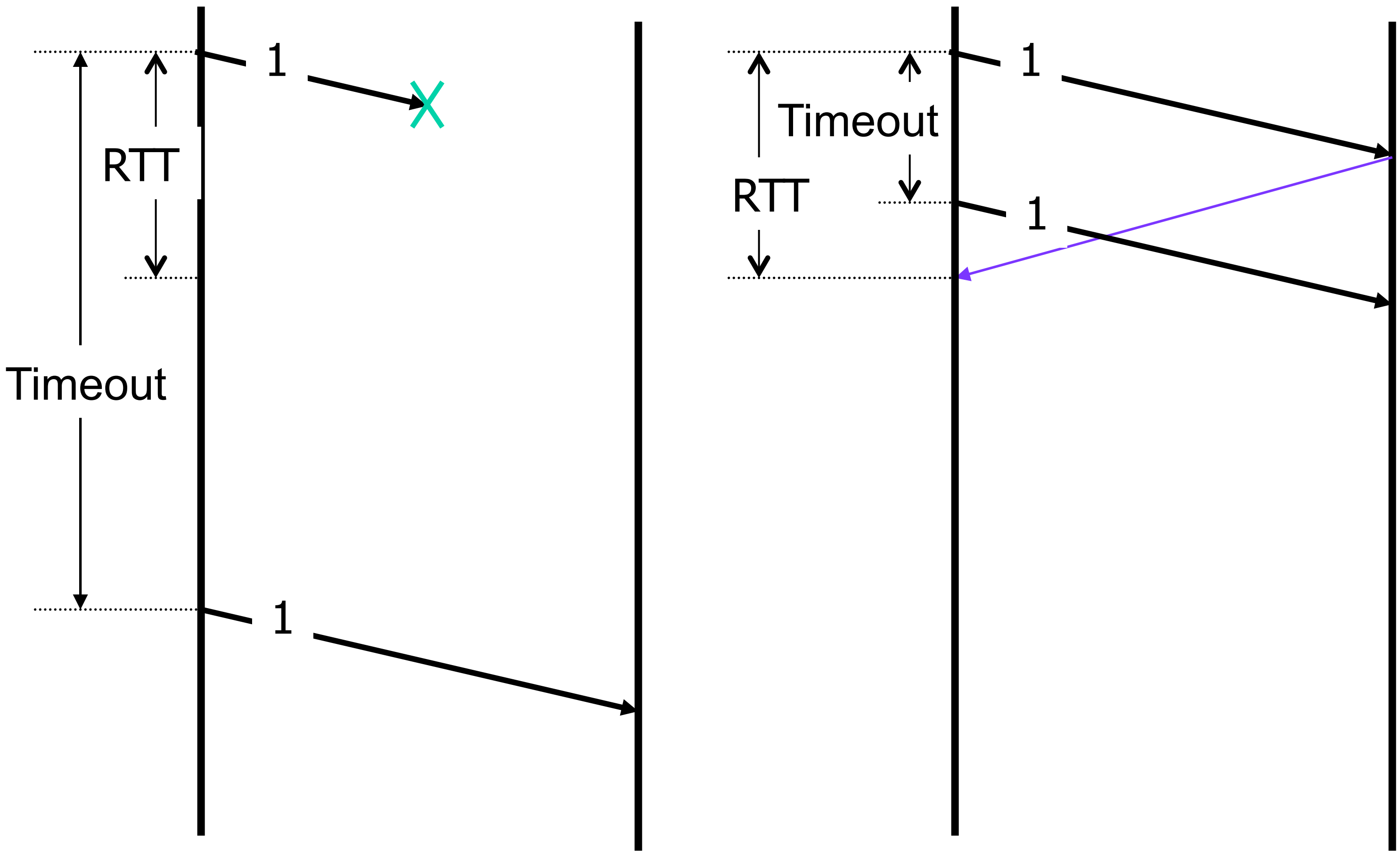
Observations

- With sliding windows, it is possible to fully utilize a link, provided the window size is large enough
  Throughput proportional to (n/RTT)
  - Stop & Wait is like n = 1
- But how to figure out the right window size to use???
  - Should be no larger than what's sustainable by the network and the sender/receiver machines
- Sender has to buffer all unacknowledged packets, because they may require retransmission
- Receiver may accept out-of-order packets within window, and delay delivery to application until the missing in-order packets are received

Setting Timeout Value

- The sender needs to pick a retransmission timeout value in order to decide when to retransmit a packet that may have been lost
- How to pick the timeout value?
  - Too short: may retransmit before data or ACK has arrived, creating duplicates
  - Too long: if a packet is lost, will take a long time to recover (inefficient)

Timing Illustration



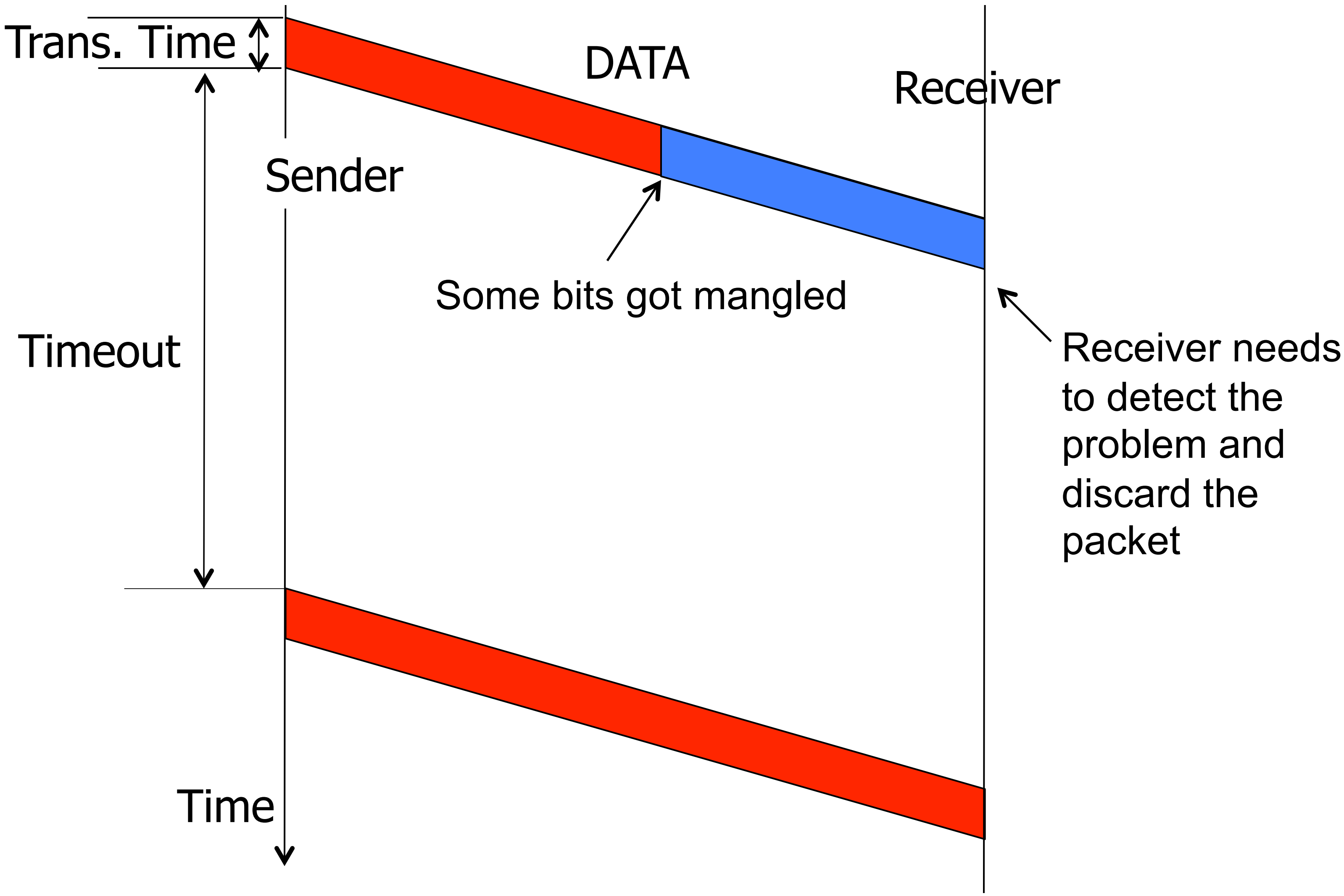
Timeout too long → inefficiency

Timeout too short → duplicate packets

Adaptive Timeout Value

- The amount of time the sender should wait is related to the round-trip time (RTT) between the sender and receiver
- RTT not known a priori and may change, so a protocol should measure RTT and adapt timeout value accordingly
- The timeout value should be larger than the RTT

Data Bit Error/Corruption



Error Detection: Naïve Approach

- Send a message twice
- Compare two copies at the receiver
  - If different, some errors exist

- How many bits of error can you detect?

- What is the overhead?

Error Detection

- Problem: detect bit errors in packets
- Solution: add extra bits to each packet
- Goals:
  - Small overhead, i.e., small number of redundancy bits
  - Large number of bit error patterns that can be detected

What is the Relationship between the red bit and the black bits?

e.g. 1001111 1

e.g. 1111011 0

e.g. 1011111 0

e.g. 1010001 1

e.g. 0100101 1

e.g. 1110111 0

Parity – Illustrates Idea Behind Error Detection

- Even parity
  - Add a parity bit to 7 bits of data to make an even number of 1's

0110100	**1**
1011010	**0**

- How many bits of error can be detected by a parity bit?
- What's the overhead?

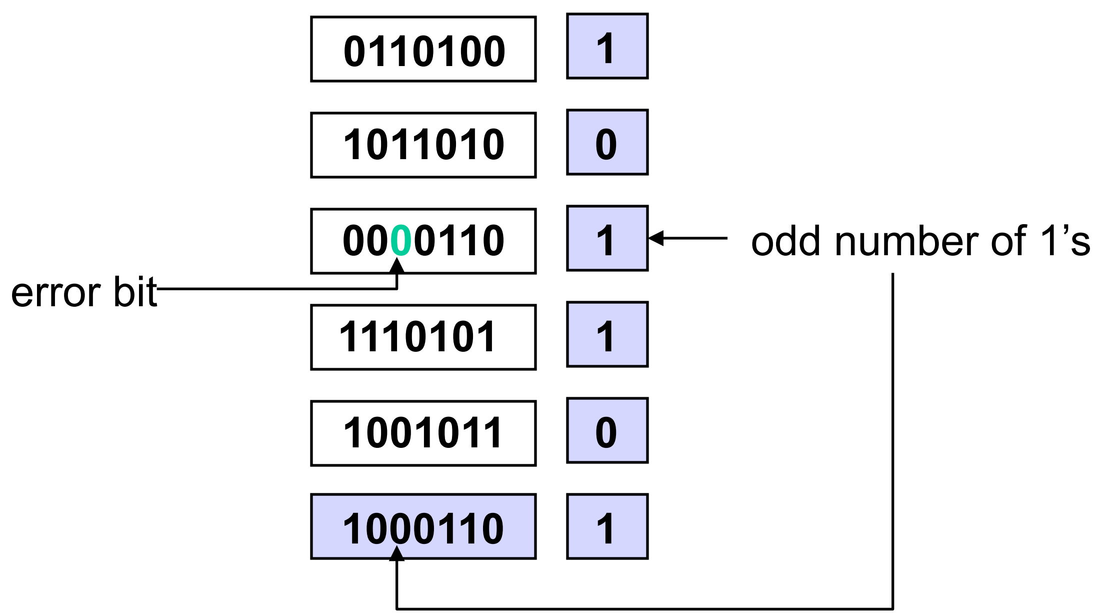
Two-dimensional Parity

- Add one extra bit to a 7-bit code such that the number of 1's in the resulting 8 bits is even (for even parity, and odd for odd parity)
- Add a parity byte for the packet
- Example: five 7-bit character packet, even parity

0110100	1
1011010	0
0010110	1
1110101	1
1001011	0
1000110	1

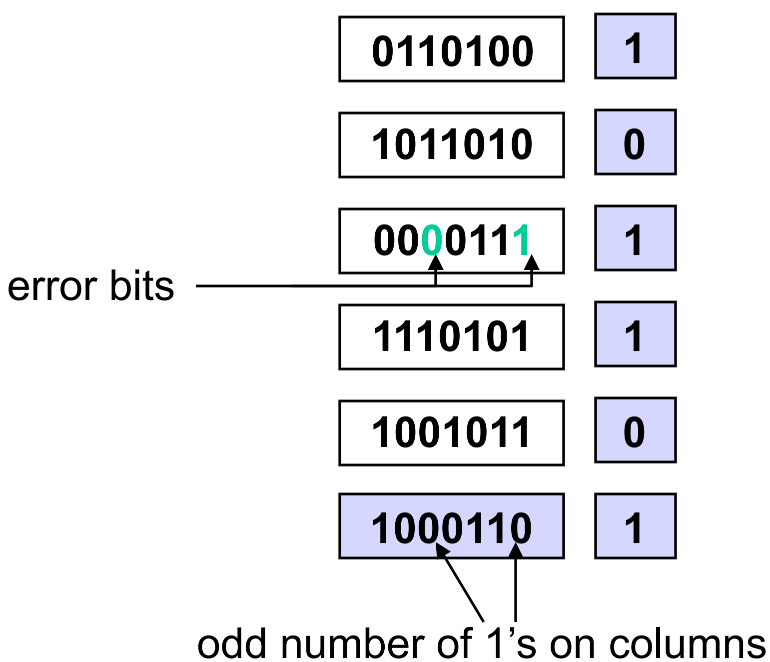
How Many Errors Can you Detect?

- All 1-bit errors
- Example:



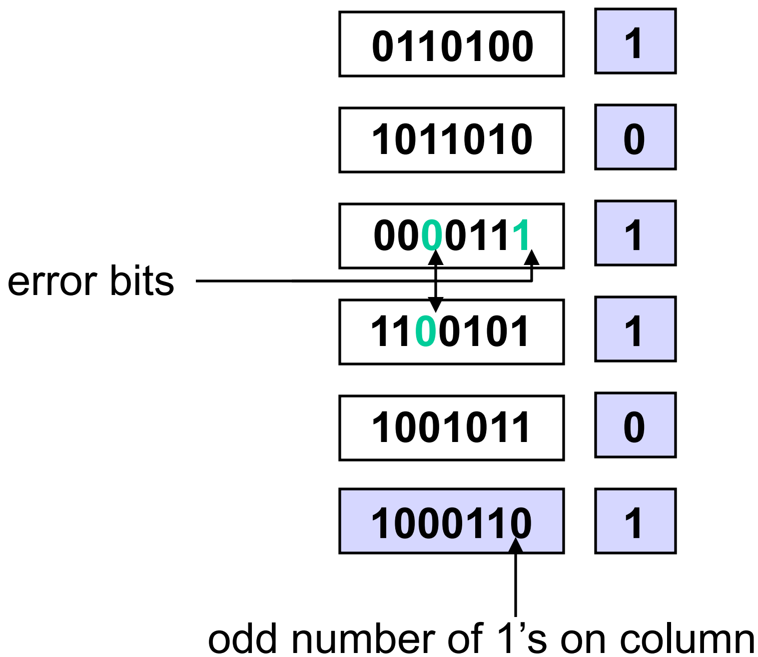
How Many Errors Can you Detect?

- All 2-bit errors
- Example:



How Many Errors Can you Detect?

- All 3-bit errors
- Example:



How Many Errors Can you Detect?

- Example of 4-bit error that is not detected:

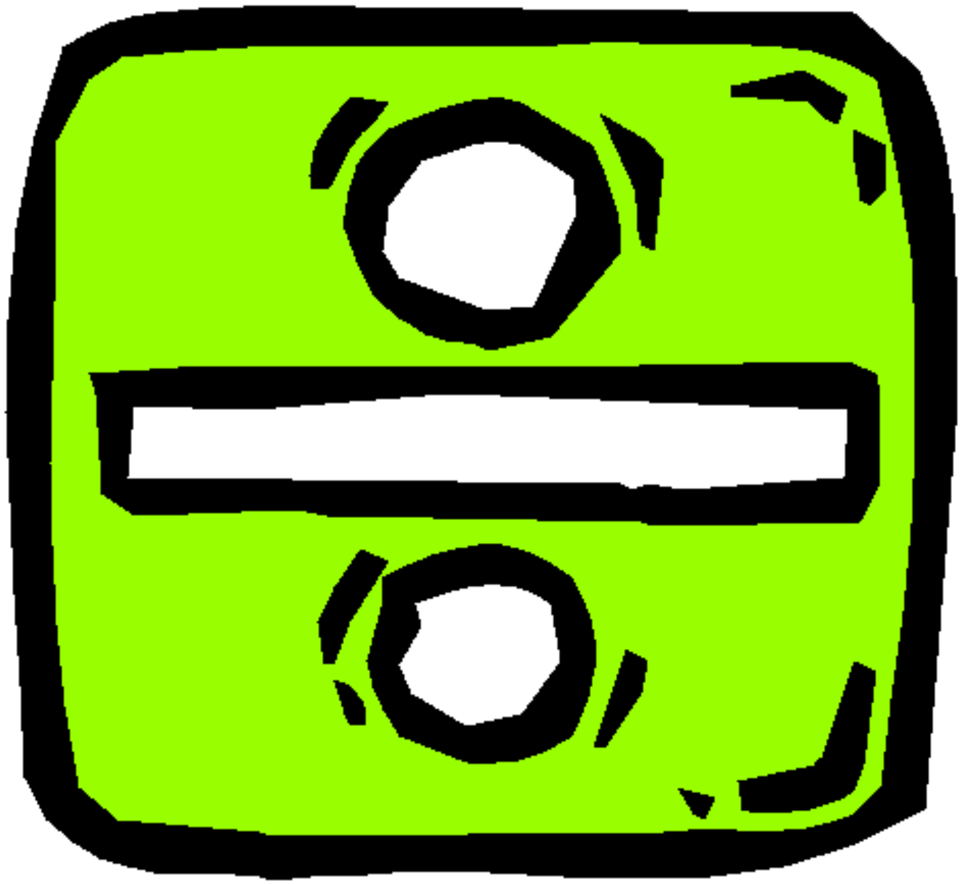
	0110100	1
	1011010	0
	0000111	1
error bits	1100100	1
	1001011	0
	1000110	1

Error Detection in Network Protocols

- Parity is used in PCI buses, cache/memory, etc.
- Parity is seldom used in network protocols

- Ethernet uses cyclic redundancy check
- Internet protocols use 1's complement sum

Cyclic Redundancy Check (CRC)



Arithmetic Modulo 2

- Like binary arithmetic but without borrowing/carrying from/to adjacent bits
- Examples:

```
101 +      101 +     1011 +          101 -     101 -     1011 -
010        001       0111            010       001       0111
----       ----      ----            ----      ----      ----
111        100       1100            111       100       1100
```

- Addition and subtraction in binary arithmetic modulo 2 is equivalent to XOR

a	b	$a \otimes b$
0	0	0
0	1	1
1	0	1
1	1	0

Some Polynomial Arithmetic Modulo 2 Properties

- Subtracting/adding C(x) from/to B(x) modulo 2 is equivalent to performing an XOR on each pair of matching coefficients of C(x) and B(x)
  - E.g.:

$$
\begin{array}{rl}
B(x) = & x^7 + x^5 + x^3 + x^2 \qquad + x^0 \quad (10101101) \\
C(x) = & x^3 \qquad + x^1 + x^0 \quad (00001011) \\
\hline
B(x) - C(x) = & x^7 + x^5 \qquad + x^2 + x^1 \qquad (10100110)
\end{array}
$$

Cyclic Redundancy Check (CRC)

- Represent a n-bit message as an (n-1) degree polynomial M(x)
  - E.g., 10101101 → $M(x) = x^7 + x^5 + x^3 + x^2 + x^0$
- Choose a divisor k-degree polynomial C(x)
- Compute remainder R(x) of $M(x)*x^k$ / C(x), i.e.,

  $\mathbf{M(x)*x^k = A(x)*C(x) + R(x)}$, where degree(R(x)) < k
- Let

  $\mathbf{T(x) = M(x)*x^k - R(x) = A(x)*C(x)}$
- Then
  - T(x) is divisible by C(x)
  - First n coefficients of T(x) represent M(x)

Cyclic Redundancy Check (CRC)

- Sender:
  - Compute and send T(x), i.e., the coefficients of T(x)
- Receiver:
  - Let T'(x) be the (n-1+k)-degree polynomial generated from the received message
  - If C(x) divides T'(x) → assume no errors; otherwise errors
- Note: all computations are modulo 2

Example (Sender Operation)

- Send packet 110111; choose C(x) = 101
  - k = 2, M(x)*x^K → 11011100
- Compute the remainder R(x) of M(x)*x^k / C(x)

```
101) 11011100
     101
      111
      101
       101
       101
         100
         101
           1  <-- R(x)
```

- Compute T(x) = M(x)*x^k - R(x) → 11011100 xor 1 = 11011101
- Send T(x)

Example (Receiver Operation)

- Assume T'(x) = 11011101
  - C(x) divides T'(x) → no errors
- Assume T'(x) = 11001101
  - Remainder R'(x) = 1 → error!

```
101) 11001101
     101
      110
      101
       111
       101
        101
        101
          1 ← R'(x)
```

- Note: an error is not detected iff C(x) divides T'(x) – T(x)

CRC Properties

- Detect all single-bit errors if coefficients of x^k and x^0 of C(x) are one
- Detect all double-bit errors, if C(x) has a factor with at least three terms
- Detect all number of odd errors, if C(x) contains factor (x+1)
- Detect all burst of errors smaller than k bits

- See Peterson & Davie Table 2.3 for some commonly used CRC polynomials

1's Complement Checksum

- Sender: add all words of data and append the result (checksum) to the data
- Receiver: add all words of received data and compare the result with the checksum

1's Complement Number Representation

- Negative number –x is x with all bits inverted
- When two numbers are added, the carry-on is added to the result
- Example: -15 + 16; assume 8-bit representation

```
15 = 00001111 → -15 = 11110000
                        +
                  16 = 00010000
                ----------------
                     [1] 00000000
                        +       1
                ----------------
                         00000001
```

-15+16 = 1

Software Implementation

```
u_short cksum(u_short *buf, int count)
{
        register u_long sum = 0;
        while (count--)
        {
                sum += *buf++;
                if (sum & 0xFFFF0000)
                {
                        /* carry occurred, so wrap around */
                        sum &= 0xFFFF;
                        sum++;
                }
        }
        return ~(sum & 0xFFFF);
}
```

Properties

- How many bits of error can 1's complement sum detect?
- What's the overhead?
- Why use this algorithm?
  - Need to understand how different technologies coordinate with each together
  - We'll start this journey in the next part of the course