Data Networks
UdS and IMPRS-CS

Lecture 12: Reliability

Overview

- Goal: transmit correct information
- Problem: bits can get corrupted
  - Electromagnetic interference, thermal noise
- Solution
  - Detect errors
  - Recover from errors
    - Correct errors (Forward error correction, FEC)
    - Retransmit corrupted data

Outline

- Error detection
- Reliable Transmission

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 (frames)
- Solution: add redundant bits to each packet
- Goals:
  - Reduce overhead, i.e., reduce the number of redundancy bits
  - Increase the number and the type of bit error patterns that can be detected
- Examples:
  - Two-dimensional parity
  - Checksum
  - Cyclic Redundancy Check (CRC)
  - Hamming Codes

Parity

- 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?

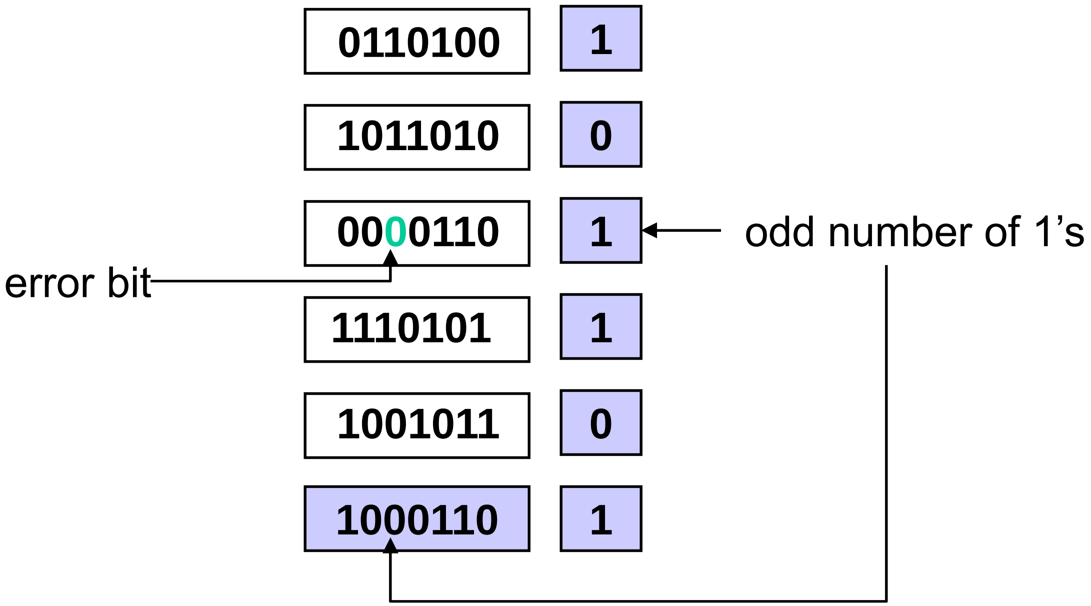
Two-dimensional Parity

- 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

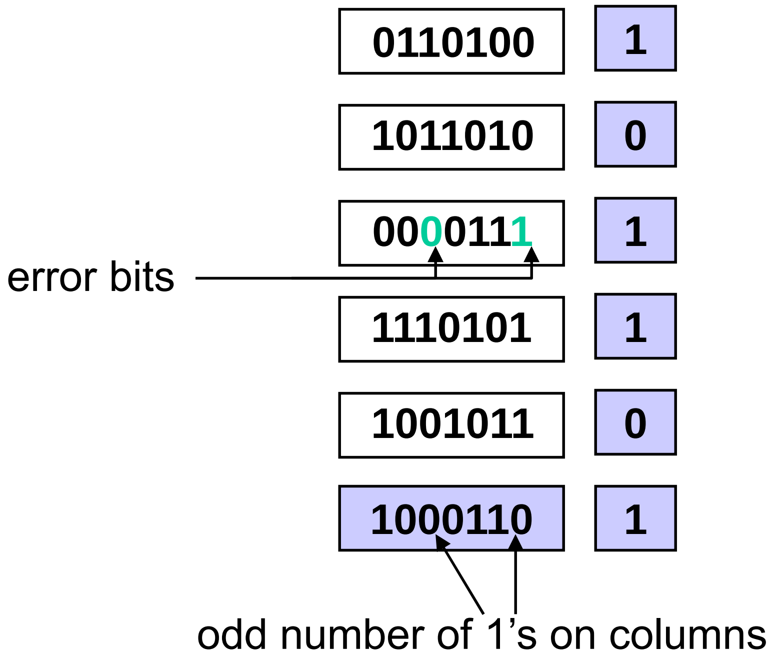
How Many Errors Can you Detect?

- All 1-bit errors
- Example:



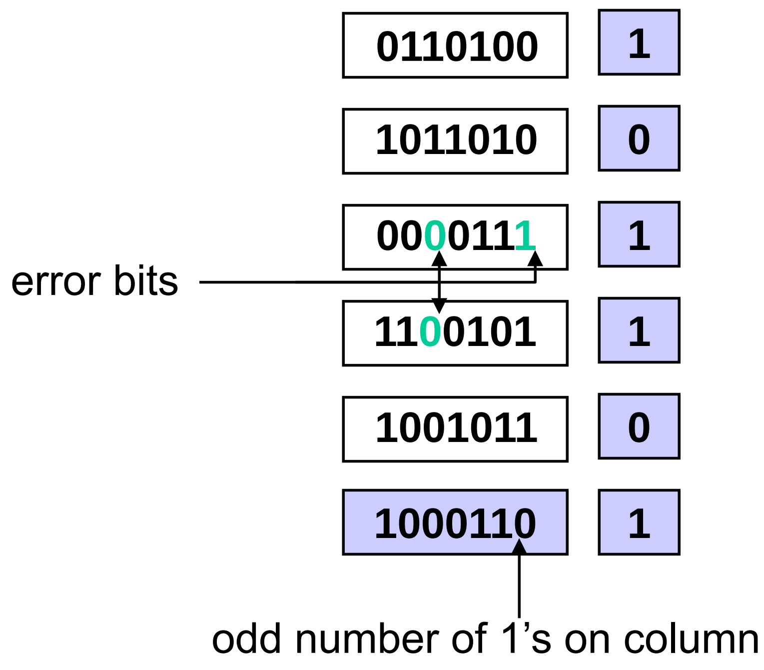
How Many Errors Can you Detect?

- All 2-bit errors
- Example:



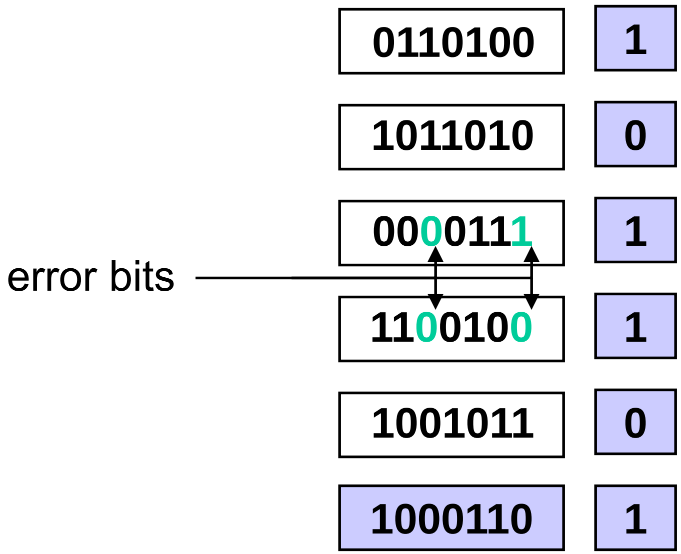
How Many Errors Can you Detect?

- All 3-bit errors
- Example:



How Many Errors Can you Detect?

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



How many errors can you correct?

Checksum

- Sender: add all words of a packet and append the result (checksum) to the packet
- Receiver: add all words of a received packet and compare the result with the checksum
- Example: Internet checksum
  - Uses 1’s complement addition

Internet Checksum 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);
}
```

1's Complement

- Negative number –x is x with all bits inverted
- When two numbers are added, any 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

Properties

- How many bits of error can the Internet checksum detect?
- What's the overhead?
- Why use this algorithm?
  - Link layer typically has stronger error detection
  - Most Internet protocol processing in the early days (70's 80's) was done in software with slow CPUs, argued for a simple algorithm
  - Applications that care about reliable delivery use stronger end-to-end detection

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 k-degree polynomial C(x)
- Compute remainder R(x) of $M(x)*x^k / C(x)$, i.e., compute A(x) such that

  $\mathbf{M(x)*x^k = A(x)*C(x) + R(x)}$, 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 (bits) of T(x) represent M(x)
  - Last k coefficients (bits) of T(x) represent R(x) (CRC check bits)

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 represented by the received message
  - If C(x) divides T'(x) → assume no errors; otherwise errors

- Note: all computations are modulo 2

Modulo 2 Arithmetic

- 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 Properties of Polynomial Arithmetic Modulo 2

- If C(x) divides B(x), then degree(B(x)) >= degree(C(x))
- 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.:

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

Example (Sender Operation)

- Send packet 110111; choose C(x) = 101
  - k = 2, $M(x)*x^K$ → 11011100
- Compute the reminder 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
  - Reminder 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 2.5 for commonly used CRC polynomials
  - CRC-32: 100000100110000010001110110110111
  - (e.g. used in Ethernet)

Overview

- Error detection
- ➢ Reliable transmission

Retransmission

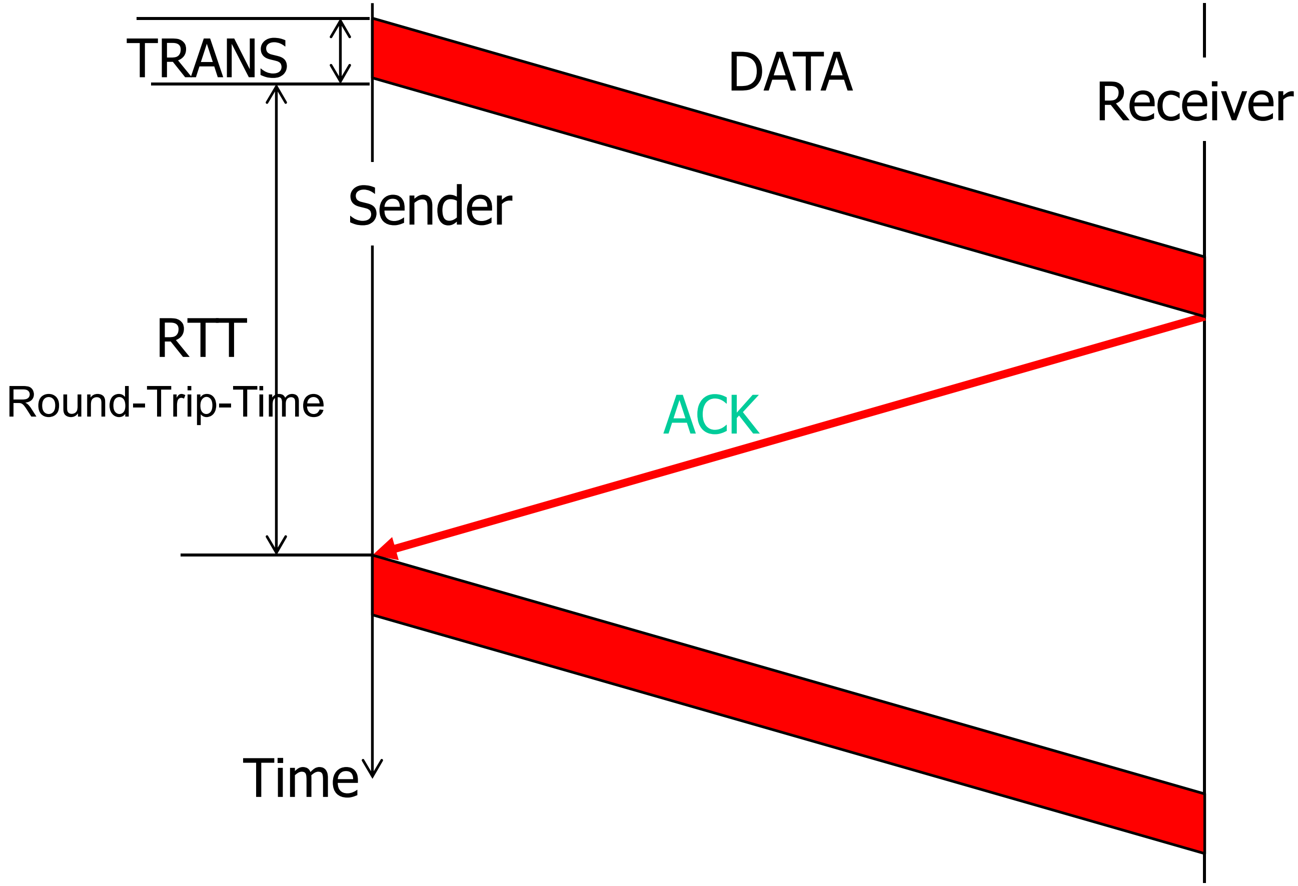
- Problem: obtain correct information once errors are detected
- Retransmission is one popular approach
- Algorithmic challenges
  - Achieve high link utilization, and low overhead

Reliable Transfer

- Retransmit missing packets
  - Numbering of packets and ACKs
- Do this efficiently
  - Keep transmitting whenever possible
  - Detect missing ACKs and retransmit quickly
- Two schemes
  - Stop & Wait
  - Sliding Window
    - Go-back-n variant
    - Selective Repeat variant

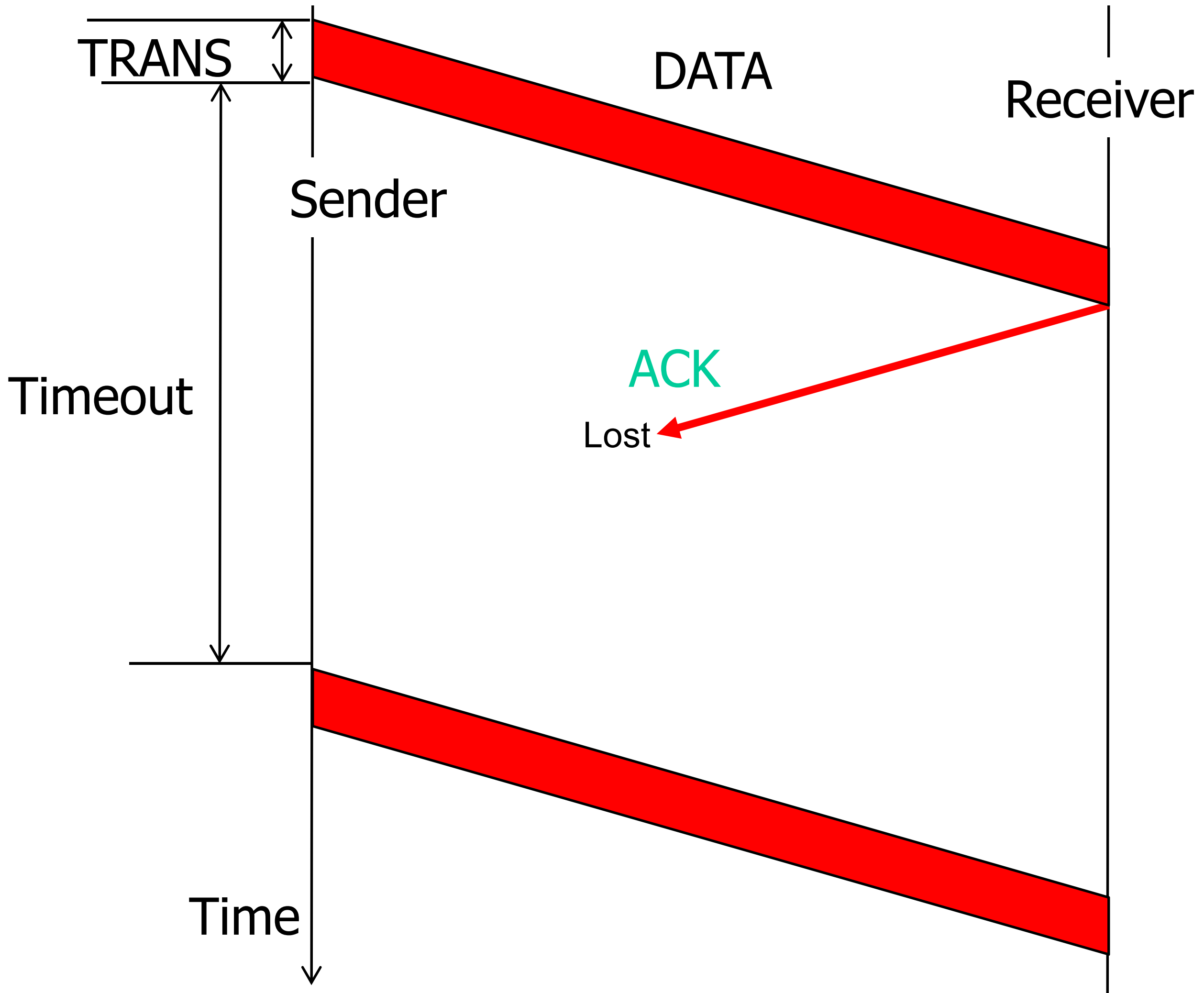
Stop & Wait

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

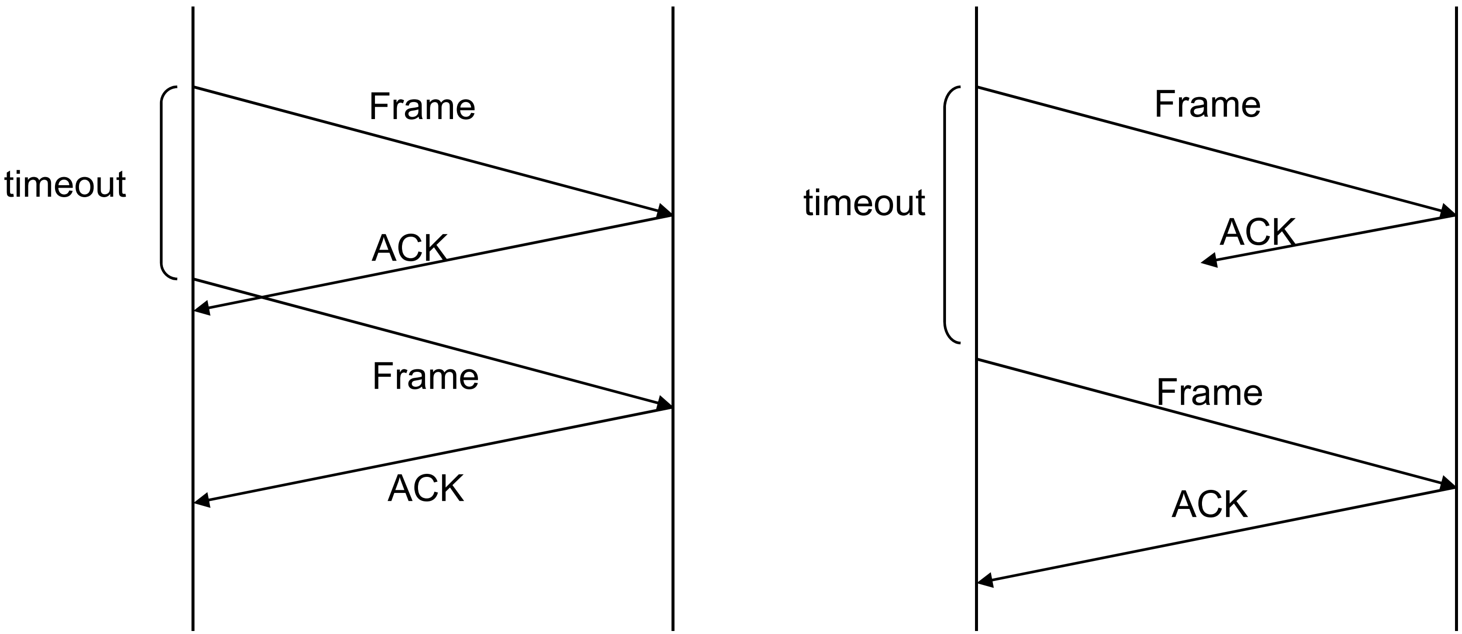


Inefficient if TRANS << RTT

Stop & Wait



Is a Sequence Number Needed?



- Need a 1 bit sequence number (i.e. alternate between 0 and 1) to detect retransmitted frames and corresponding ACKs

Problem with Stop-and-Go

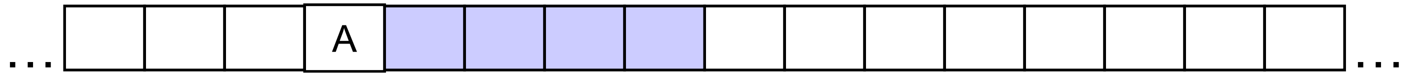
- Lots of time wasted in waiting for acknowledgements
- What if you have a 10Gbps link and a delay of 10ms?
  - Need 100Mbit to fill the pipe with data
- If packet size is 1500B (like Ethernet)
  - can only send one packet per RTT
  - Throughput = 1500*8bit/(2*10ms) = 600Kbps!
  - A utilization of 0.006%
- In general, U = D / D + H + A + 2CI

(Utilization U, D data bits, H header bits, A ACK bits, Channel capacity C, propagation delay I)

Sliding Window

- *Window* = set of adjacent sequence numbers
- The size of the set *n* is the *window size (WS)*

- Let A be the last ACKed packet of sender without gap; then window of sender = {A+1, A+2, …, A+n}
  - Sender window size (SWS)

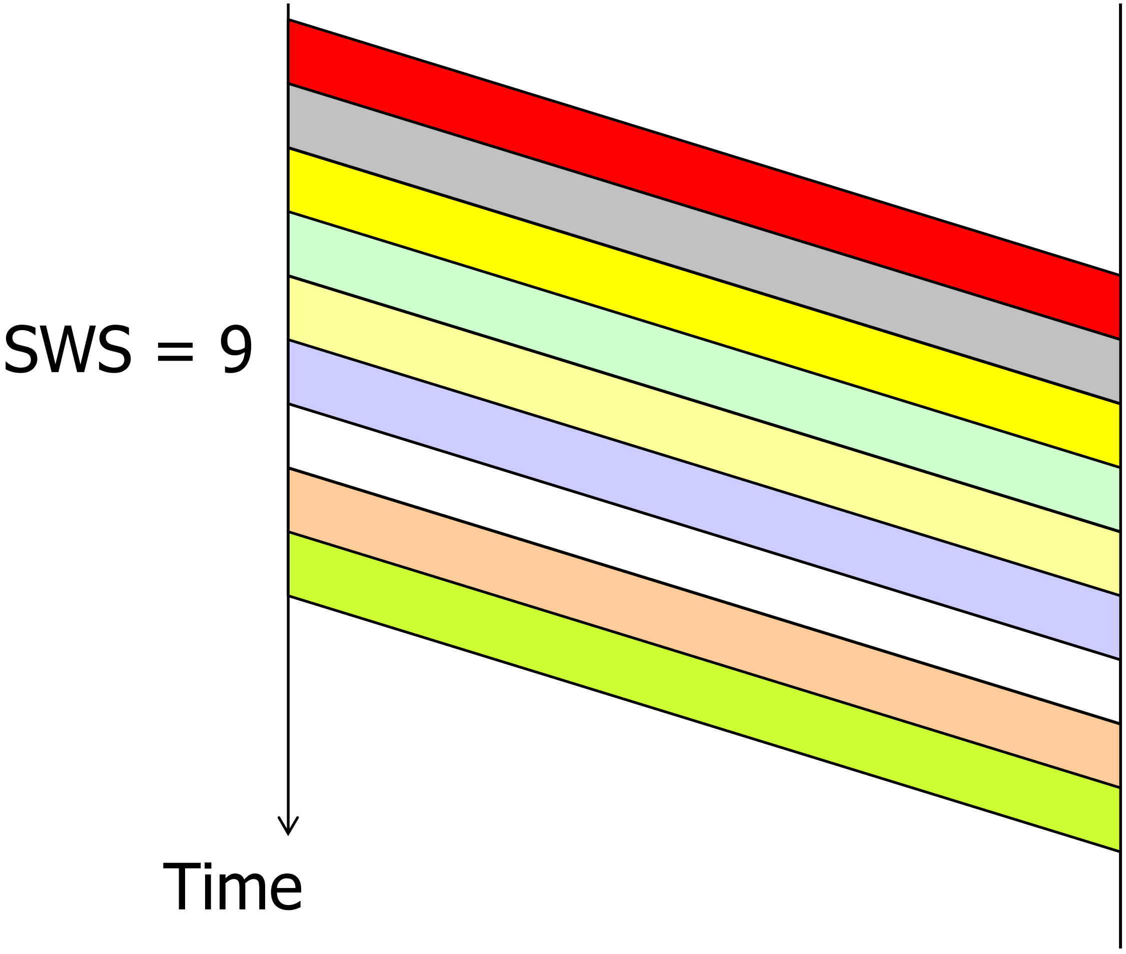


- Sender can send packets in its window

- Let B be the last received packet without gap by receiver, then window of receiver = {B+1,…, B+n}
  - Receiver window size (RWS)

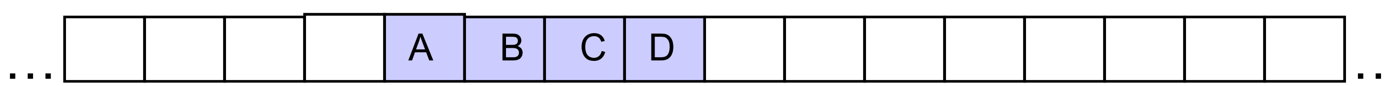
- Receiver can accept out of sequence packets, if in window

Example



Basic Timeout and Acknowledgement

- Every packet k transmitted is associated with a timeout
- If by timeout(k), the ACK for k has not yet been received, the sender retransmits k
- Basic acknowledgement scheme
  - Receiver sends ACK for packet k when all packets with sequence numbers <= k have been received
  - An ACK k means every packet up to k has been received



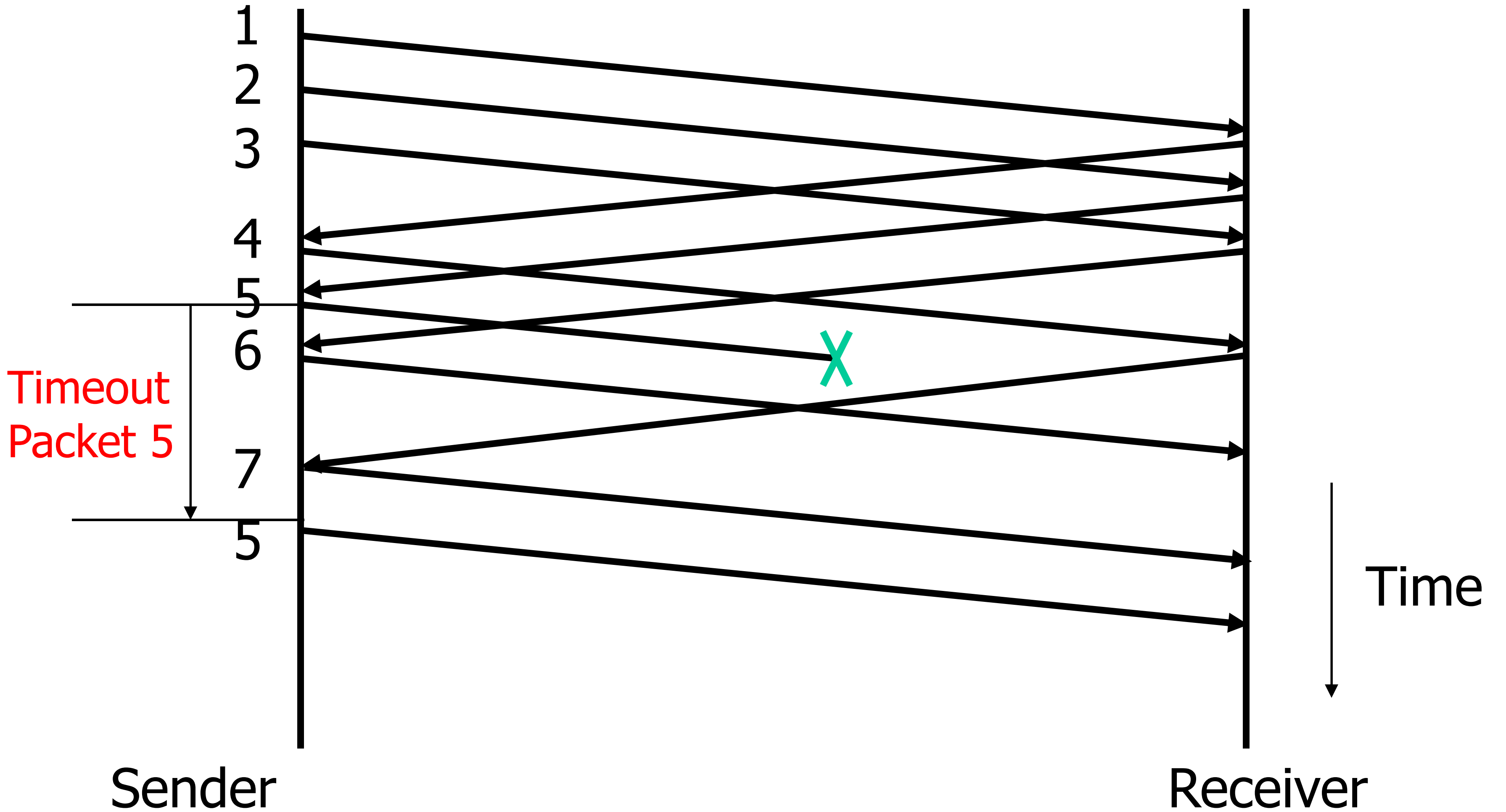
  - Suppose packets B, C, D have been received, but receiver is still waiting for A. No ACK is sent when receiving B,C,D. But as soon as A arrives, an ACK for D is sent by the receiver, and the receiver window slides

Error recovery strategies

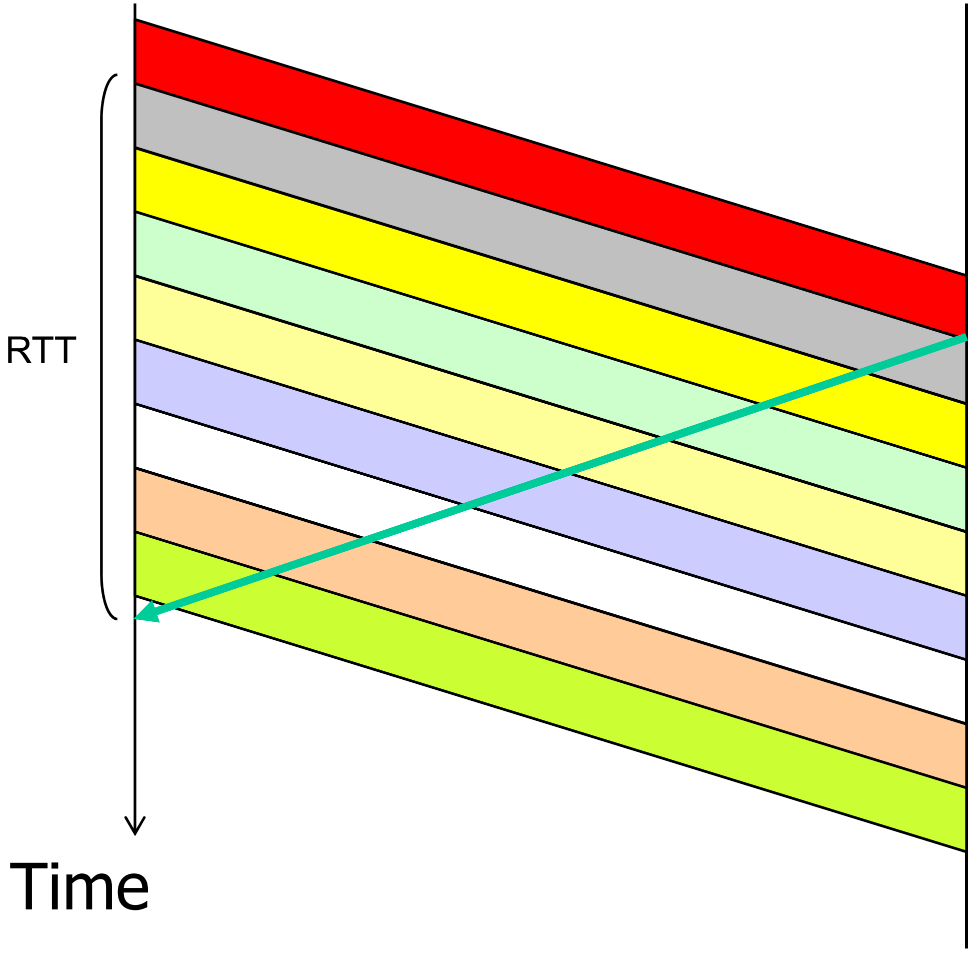
- Solution 1: Go Back N
  - Receiver discards all subsequent frames (RWS=1)
  - Errors disrupt transmission pipeline
  - Bandwidth wasted when error rate is high
- Solution 2: Selective retransmission
  - Receiver may or may not send NAK to identify missing frames explicitly (duplicate ACKs indicate missing frames!)
  - Receiver stores subsequent frames (RWS=SWS)
  - Sender retransmits missing frame(s) only
- Many variations

Example with Errors

Window size = 3 packets



Efficiency



SWS = 9, i.e. 9 packets per RTT instead of 1

→ Can saturate link if window is large enough

SWS ≥ 2CI / (H + D) + 1
→ U = D / H + D

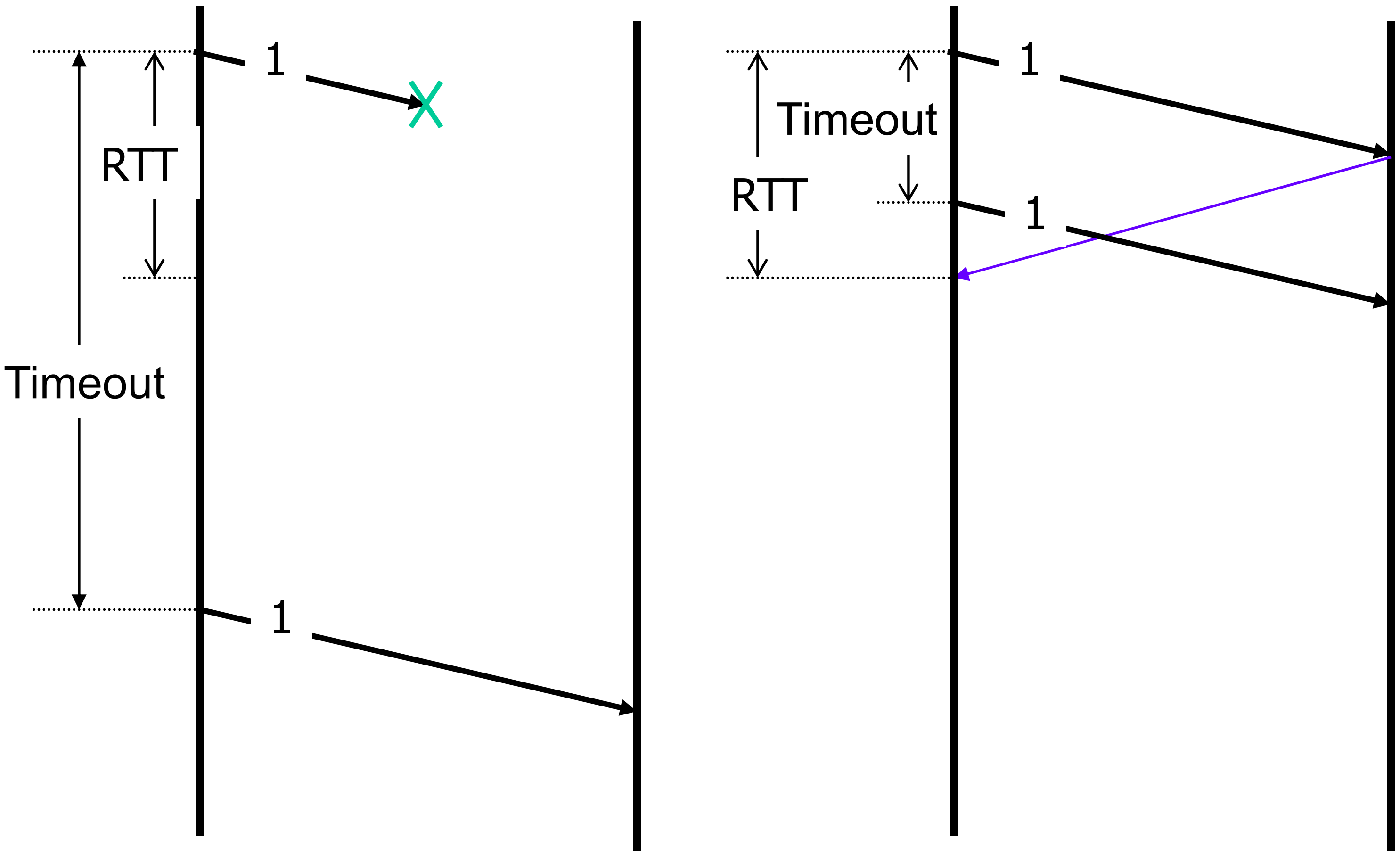
Observations

- With sliding windows, it is possible to fully utilize a link, provided the window size is large enough. Throughput is ~ (n/RTT)
  - Stop & Wait is like n = 1.
- Sender has to buffer all unacknowledged packets, because they may require retransmission
- Receiver may be able to accept out-of-order packets, but only up to its buffer limits
- Need sequence numbers with range ≥ 2 SWS

Setting Timers

- The sender needs to set retransmission timers in order to know when to retransmit a packet that may have been lost
- How long to set the timer for?
  - 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 Timers

- The amount of time the sender should wait is about the round-trip time (RTT) between the sender and receiver
- For link-layer networks (LANs), this value is essentially known
- For multi-hop networks, rarely known (queuing!)
- Should work in both environments, so protocol should adapt to the path behavior
- E.g. TCP timeouts are adaptive, will discuss later in the course