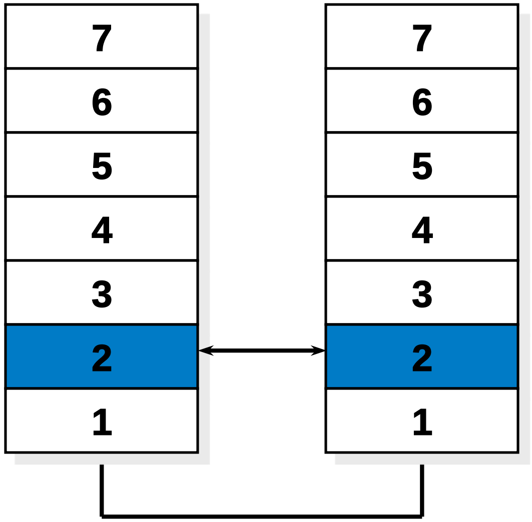
Operating Systems & Computer Networks

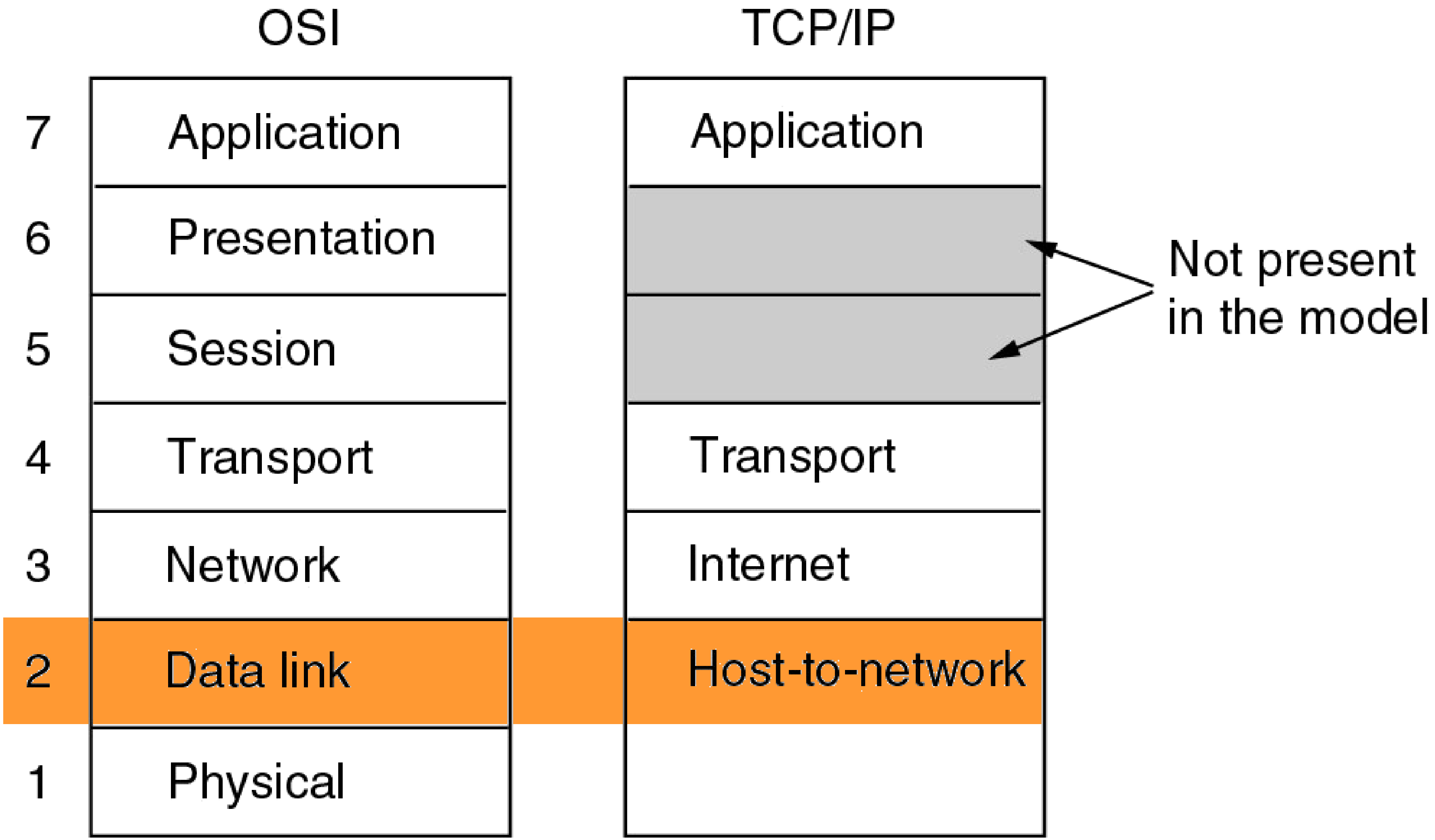
Host-to-Network II

- Data Link Layer
- Framing, Flow Control
- Error Detection / Correction
- Point-to-Point Protocol



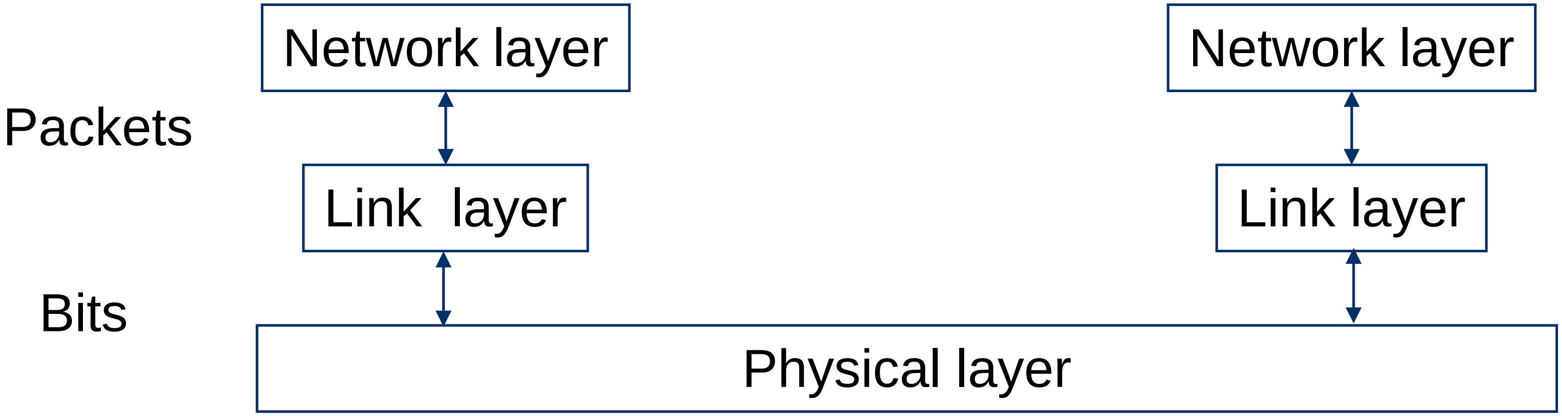
Content (2)

Data Link Layer



Setting for Data Link Layer

- Link layer sits on top of physical layer
  - Can thus use a bit stream transmission service
  - But: Service might have incorrect bits
- Expectations of the higher layer (networking layer)
  - Wants to use either a packet service
    - Rarely, a bit stream service
  - Does not really want to be bothered by errors
  - Does not really want to care about issues at the other end



Services of Data Link Layer

Given setting and goals, the following services are required:

- Transparent communication between two directly connected nodes
- Framing of a physical bit stream into a structure of frames/packets
  - Frames can be retransmitted, scheduled, ordered, ...
- Error control
  - Detection and correction
- Connection setup and release
  - Signaling and resource management on hosts
- Acknowledgement-based protocols
  - Make sure that a frame has been transmitted
- Flow control
  - Arrange for appropriate transmission speed between hosts

Framing

Link Layer Functions – Framing

- How to turn bit stream abstraction (as provided by physical layer) into individual, well demarcated *frames*?
  - Usually necessary to provide error control
    - Not obvious how to do that over a bit stream abstraction
  - Frames and packets are really the same thing
    - Term “frames” used in link layer context by convention

- Additionally: Fragmentation and reassembly
  - If network layer packets are longer than link layer packets
  - ➢ Commonly network layer set Maximum Transmission Unit (MTU) to a value that avoids link layer fragmentation
    - 1492 bytes on 802.3 (Ethernet)
    - 2272 bytes on 802.11 (WLAN)

Framing

- How to turn a bit stream into a sequence of frames?
  - More precisely: how does a receiver know when a frame starts and when it ends?

Delivered by physical layer:

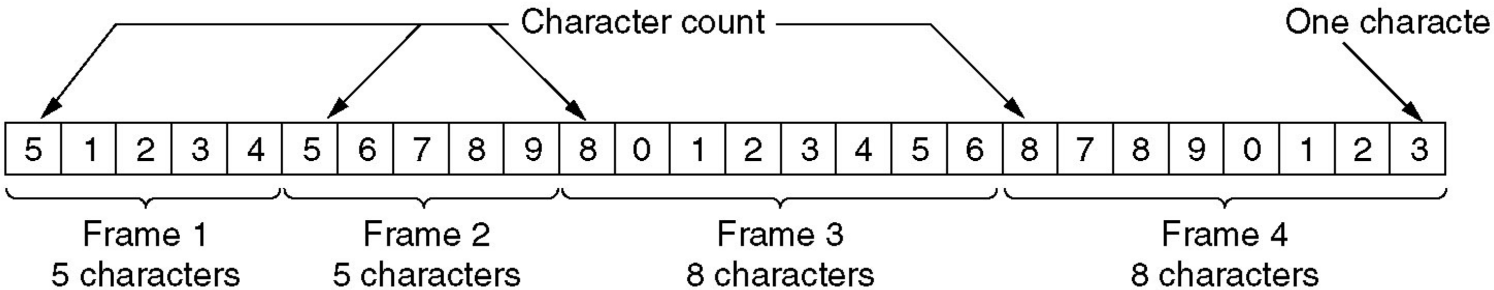
0110010101110101110010100010101010101010101100010

Start of frame (?)   End of frame (?)

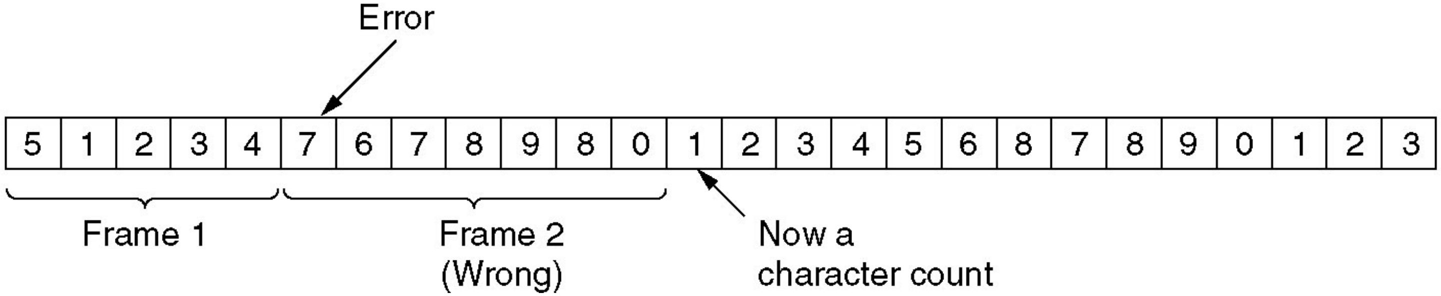
- Note: Physical layer might try to detect and deliver bits when the sender is not actually transmitting anything
  - Receiver tries to get any information from the physical medium

Framing by Character / Byte Count

- Idea: Announce number of bits (bytes, characters) in a frame to the receiver
  - Put this information at beginning of a frame – *frame header*

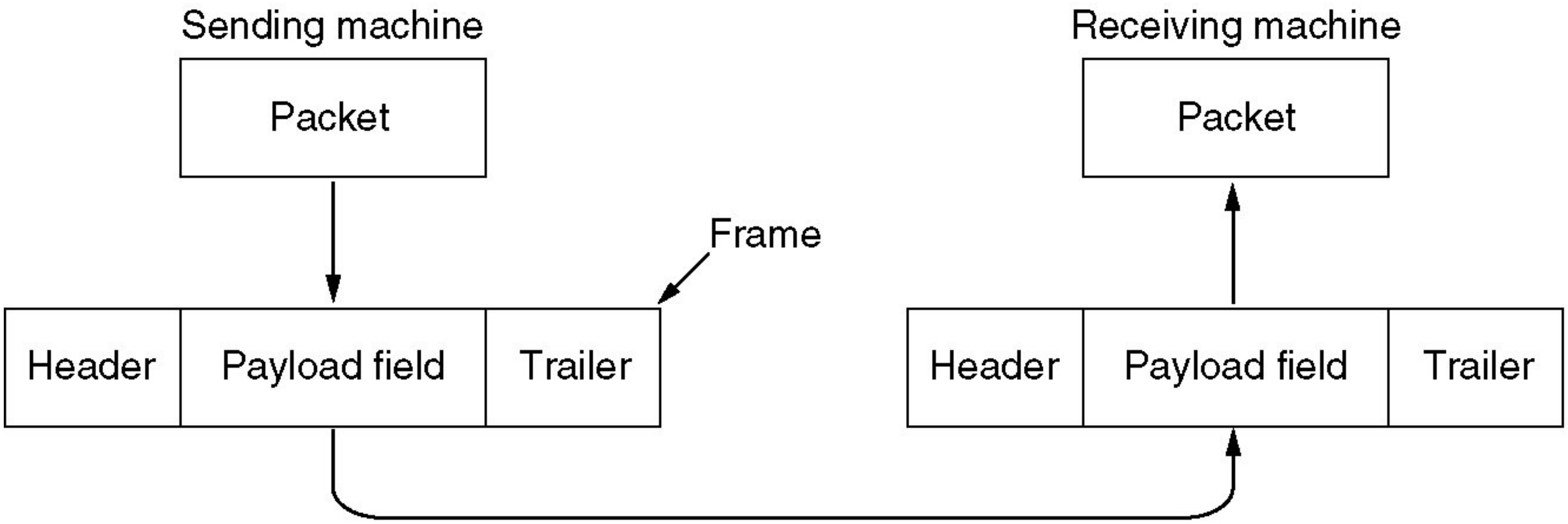


- Problem: What happens if *count* information itself is damaged during transmission?
  - Receiver will loose frame synchronization and produce different sequence of frames than original one



Basic Technique: Control Header

- Albeit “character count” is not a good framing technique, it illustrates an important technique: *headers*
  - If sender has to communicate administrative or control data to receiver, it can be added to actual packet content (“payload”)
  - Usually at the start of the packet; sometimes at the end (“trailer”)
  - Receiver uses headers to learn about sender’s intention
  - ➢ Same principle applicable to all packet-switched communication

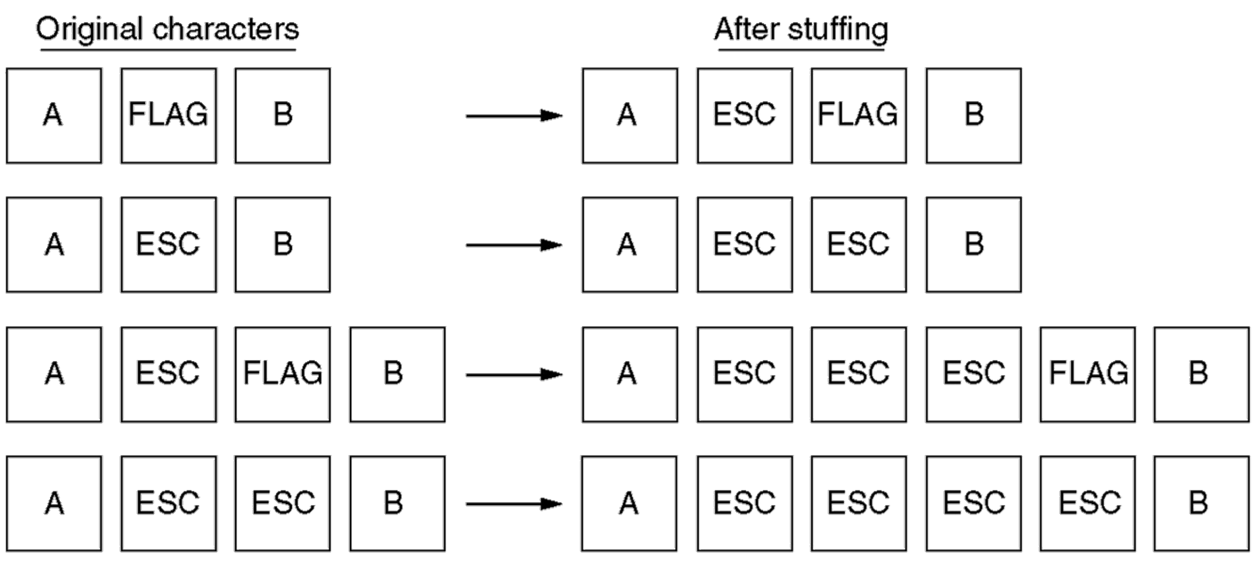


Framing by Flag Bytes / Byte Stuffing

- Use dedicated flag bytes to demarcate start/stop of frame

FLAG	Header	Payload field	Trailer	FLAG

➢ What happens if the flag byte appears in the payload?
  - Escape it with a special control character – *byte stuffing*
  - If that appears, escape it as well



Framing by Flag Bit Patterns / Bit Stuffing

- Byte stuffing is closely tied to characters/bytes as fundamental unit – often not appropriate

➢ Use same idea, but stick with the bit stream abstraction of the physical layer
  - Use bit pattern instead of flag byte – often, 01111110
    - Actually, it IS a flag byte
  - Bit stuffing process:
    - Whenever sender sends five 1s in a row, it automatically adds a 0 into bit stream – except in flag pattern
    - Receiver throws away (“destuffs”) any 0 after five 1s

Original payload (a) 0 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 1 0

After bit stuffing (b) 0 1 1 0 1 1 1 1 1 0 1 1 1 1 1 0 1 1 1 1 1 0 1 0 0 1 0

Stuffed bits

After destuffing (c) 0 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 1 0

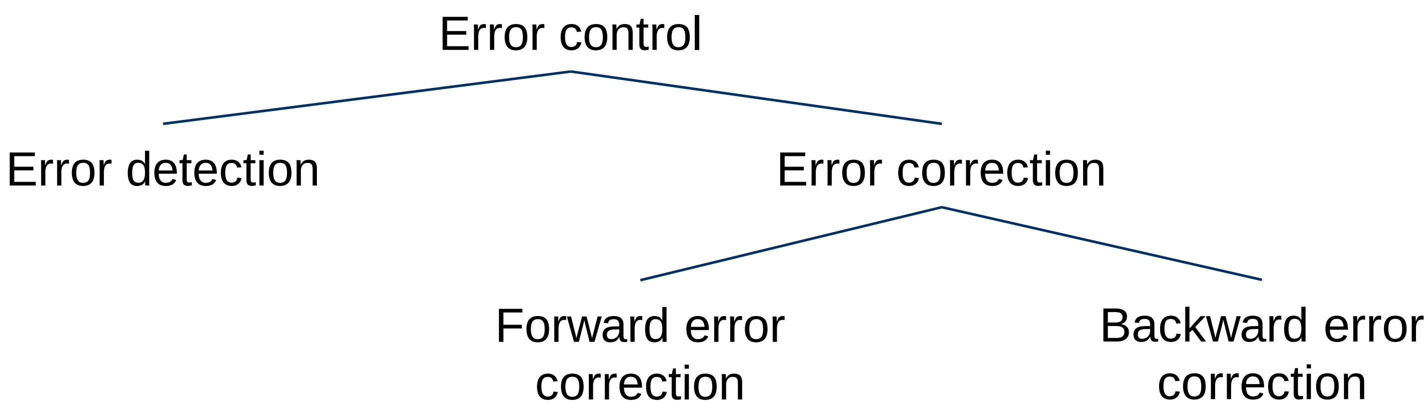
Framing by Coding Violations

- Suppose the physical layer's encoding rules "bits ! signals" still provide some options to play with
  - Not all possible combinations that physical layer can represent are used to represent bit patterns
  - ➢ Example: Manchester encoding: only low/high and high/low is used
- When "violating" these encoding rules, data can be transmitted, e.g., start and end of frame
  - ➢ Example: Manchester – use high/high or low/low
    - This drops self-clocking feature of Manchester, but clock synchronization is sufficiently good to hold for a short while
- Powerful and simple scheme, e.g. used by Ethernet networks
  - But raises questions regarding bandwidth efficiency as coding is *obviously* not optimal

Error Control

Link Layer Functions – Error Control

- Error detection – Check for incorrect bits
- Error correction – Correct erroneous bits
  - Forward error correction (FEC) – invest effort *before* error happened; avoid delays in dealing with it
    - ➢ Redundancy / overhead
  - Backward error correction – invest effort *after* error happened; try to repair it ➔ ARQ (Automatic Repeat reQuest)
    - ➢ Delays



➢ Usually build on top of framing

Error Detection:
Cyclic Redundancy Check (CRC)

- CRC can check arbitrary, unstructured sequence of bits
- A check sequence (CRC code) is appended to checked data
  - Typically calculated by a feedback shift register in hardware

Error checked data	CRC

- When calculating the CRC code a generator polynomial is used which is known to both sender and receiver

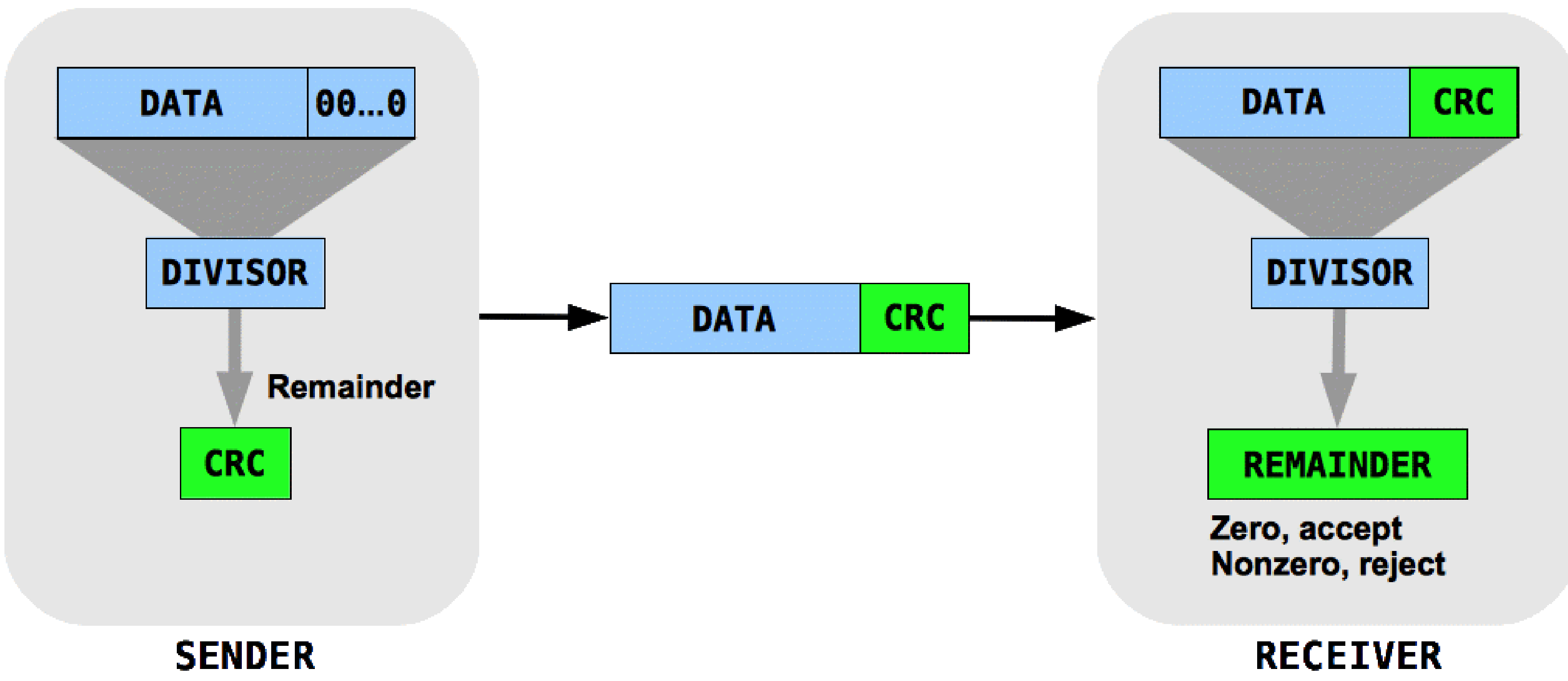
- Calculation of CRC code
  1. View bit sequence as polynomial with binary coefficients:
     - **100010** is viewed as $\mathbf{1}*x^5+\mathbf{0}*x^4+\mathbf{0}*x^3+\mathbf{0}*x^2+\mathbf{1}*x^1+\mathbf{0}*x^0$
  2. Expand polynomial with n 0s, n is the degree of the generator polynomial
  3. Divide expanded bit sequence (i.e. polynomial) by generator polynomial

  ➢ CRC code is the remainder of the division, result is discarded

  4. Receiver again divides received bit sequence (including the CRC code) by generator polynomial

  ➢ If no error occurred the remainder is 0

Illustration of CRC



CRC Example (Sender)

- Transmitted payload: 110011
- Generator polynomial: $x^4 + x^3 + 1$
  - ➢ Translates into sequence of coefficients: 11001
  - Addition or subtraction equal simple bitwise XOR
    - ➢ Special arithmetic for polynomials modulo 2
- Length of CRC = degree of generator polynomial = 4
- Calculation of CRC:

```
1100110000 ÷ 11001 = 100001 (mod 2)
11001
0000010000
     11001
      1001  = remainder
```

➢ Transmitted bit sequence: 1100111001

CRC Example (Receiver)

- Reception of a correct bit sequence:

```
1100111001 ÷ 11001 = 100001 (mod 2)
11001
0000011001
     11001
     00000 = remainder
```

  - ➢ No remainder, thus the received bits *should* be error free

- Reception of a erroneous bit sequence:

```
1111111000 ÷ 11001 = 101001 (mod 2)
11001
0011011
  11001
  00010000
     11001
     01001 = remainder ≠ 0
```

  - ➢ There is a remainder unequal 0, thus there was *definitely* a transmission error

Properties of CRC

CRC can detect the following errors:

- All single bit errors
- All double bit errors (if $(x^k + 1)$ is not divisible by generator polynomial for $k \leq$ frame length)
- All errors affecting an odd number of bits (if $(x+1)$ is a factor of the generator polynomial)
- All error bursts of length $\leq$ degree of generator polynomial

Internationally standardized generator polynomials:

- CRC-12 $= x^{12} + x^{11} + x^3 + x^2 + x + 1$
- CRC-16 $= x^{16} + x^{15} + x^2 + 1$
- CRC-CCITT $= x^{16} + x^{12} + x^5 + 1$

CRC-16 and CRC-CCITT detect

- All single and double errors
- All errors affecting an odd number of bits
- All error bursts of length ≤ 16
- 99,997% of all error bursts of length 17
- 99,998% of all error bursts of length ≥ 18

FEC Example (Sender)

- CRC uses redundancy to *detect* errors
- FEC uses redundancy to *correct* errors

- Simple FEC scheme: Repeat data several times, then use majority decision
  - ➢ Problematic with regard to overhead
- More advanced: XOR, Reed-Solomon, ...

- Example (XOR):
  - To send the following three packets:

    0101 - P1
    1111 - P2
    0000 - P3

1. Calculate a fourth packet using XOR:

   1010 - P4

2. Send all four packets to the receiver
   - ➢ Overhead: (4 – 3) / 4 = 25%

FEC Example (Receiver)

- For the receiver, it is sufficient to receive any three out of the four packets.

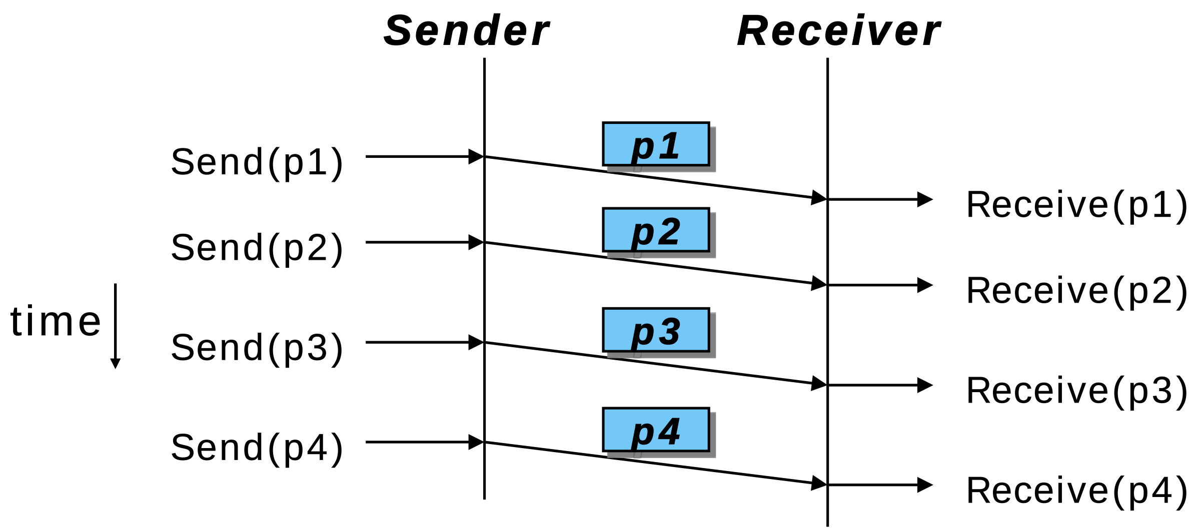
- Reconstruct missing packet based on received packets:
    - P1 is lost: 1111
0000
1010
0101 -> P1
    - P2 is lost: 0101
0000
1010
1111 -> P2
    - P3 is lost: 0101
1111
1010
0000 -> P3

- However, receiver still has to know which packet was lost
    - Requires packet numbering

Flow Control

Link Layer Functions – Flow Control

- Assumptions in an ideal world:
  - Sender/receiver are always ready to send/receive
  - Receiver can handle amount of incoming data
  - No errors occur that cannot be handled by FEC



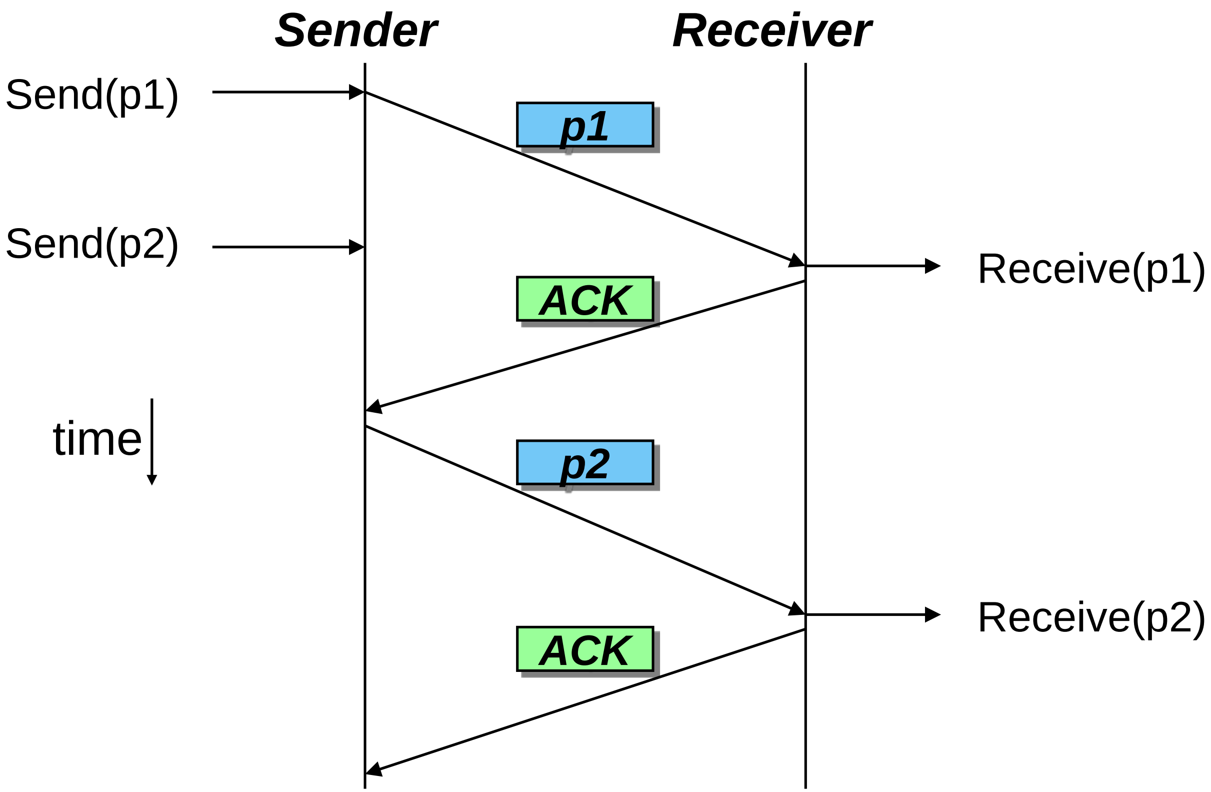
- ➢ What happens if packets are lost?
- ➢ What happens if the sender floods the receiver?
  - Imagine a web server sending data to a mobile phone…

Very Simple Solution: Stop-and-Wait

- Concentrate on one single packet
- Receiver acknowledges correct reception of the packet
- Sender has to wait for that acknowledgement before continuing with next packet

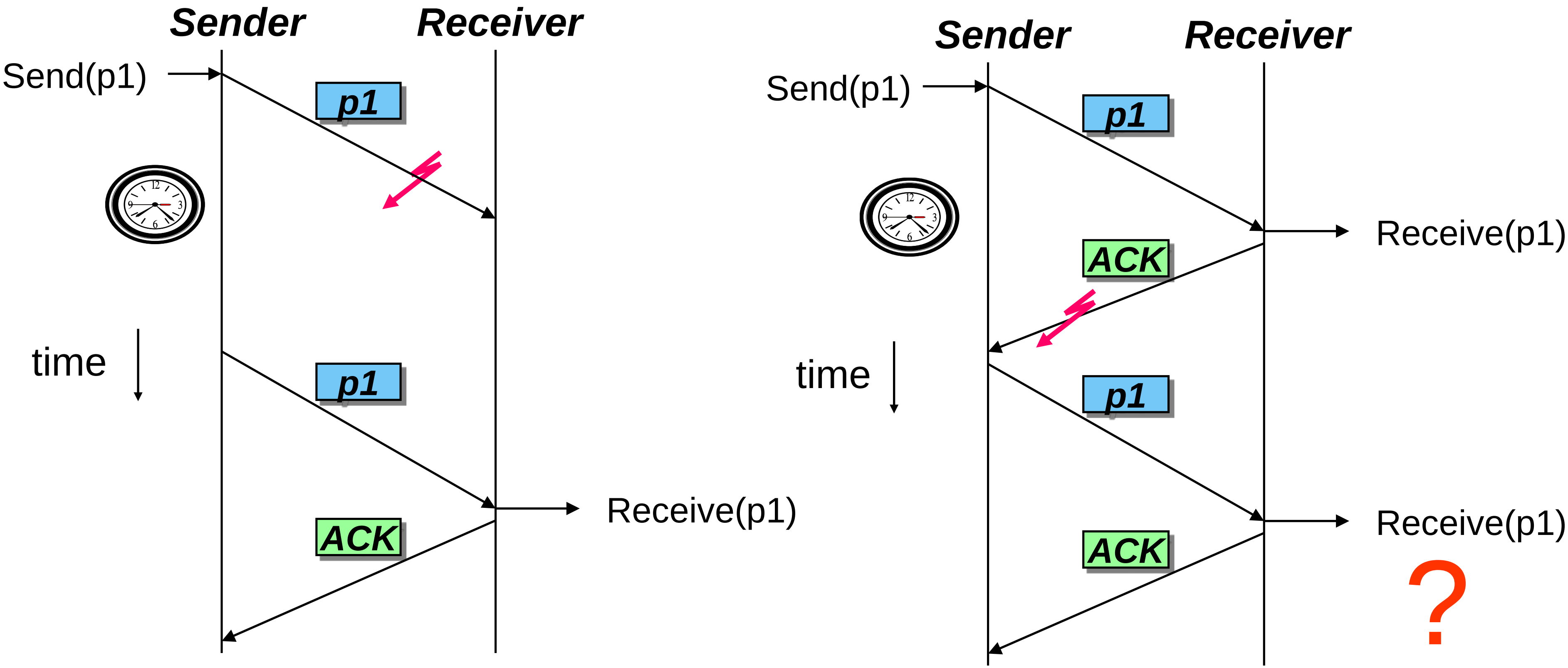
- No overloading of the receiver possible!

➢ Basic flow control



Problems of Stop-and-Wait

- What happens if errors occur?
  - Lost packet? Lost acknowledgement? Is there a difference?
- Basic solution: ARQ (Automatic Repeat reQuest)



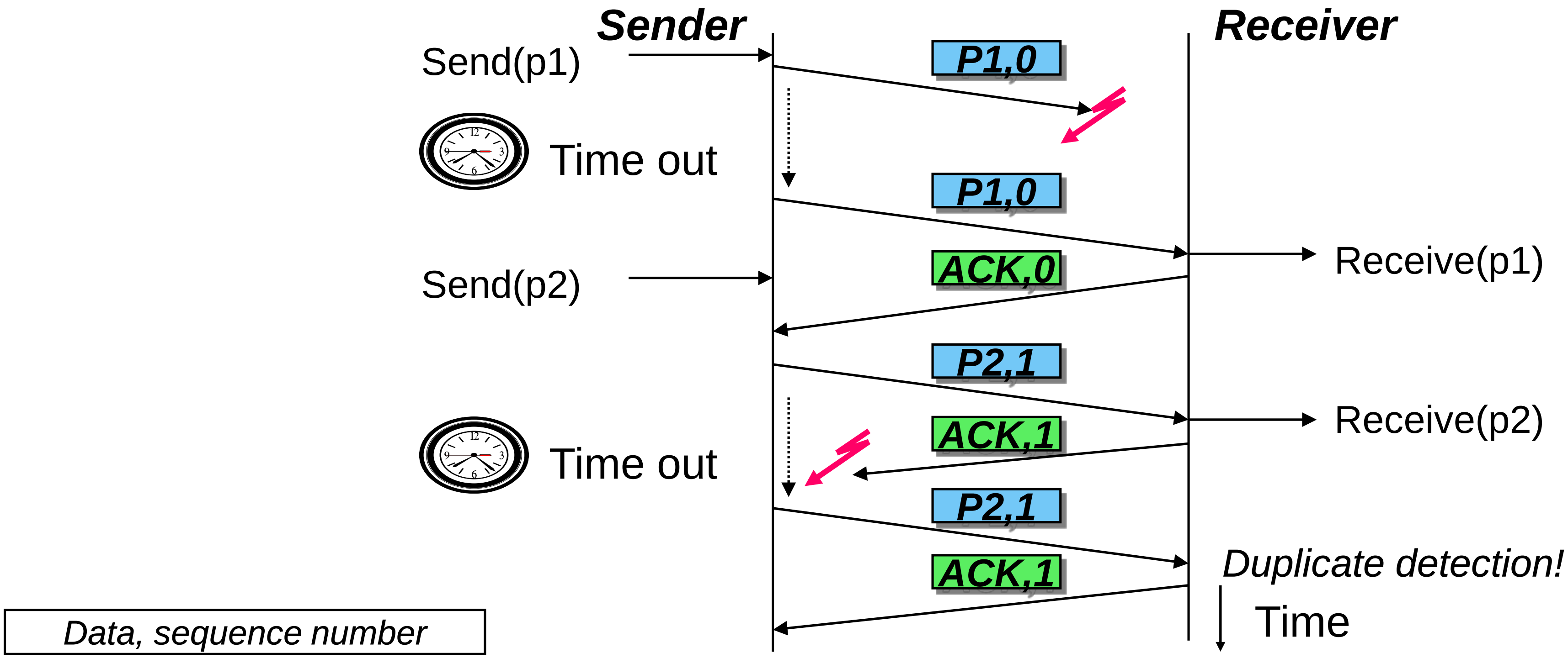
Problem of Stop-and-Wait ARQ

- Sender cannot distinguish between lost packet and lost acknowledgement ➔ Has to re-send the packet
- Receiver cannot distinguish between new packet and redundant copy of old packet ➔ Additional information is needed

➢ Put a *sequence number* in each packet, telling the receiver which packet it is
  - Sequence numbers as *header information* in each packet
  - Simplest sequence number: 0 or 1
- Needed in packet and acknowledgement
  - One convention: In ACK, send sequence number of last correctly received packet
  - Also possible: Send sequence number of next expected packet
  - ➢ Be aware: Some protocols count bytes instead of packets

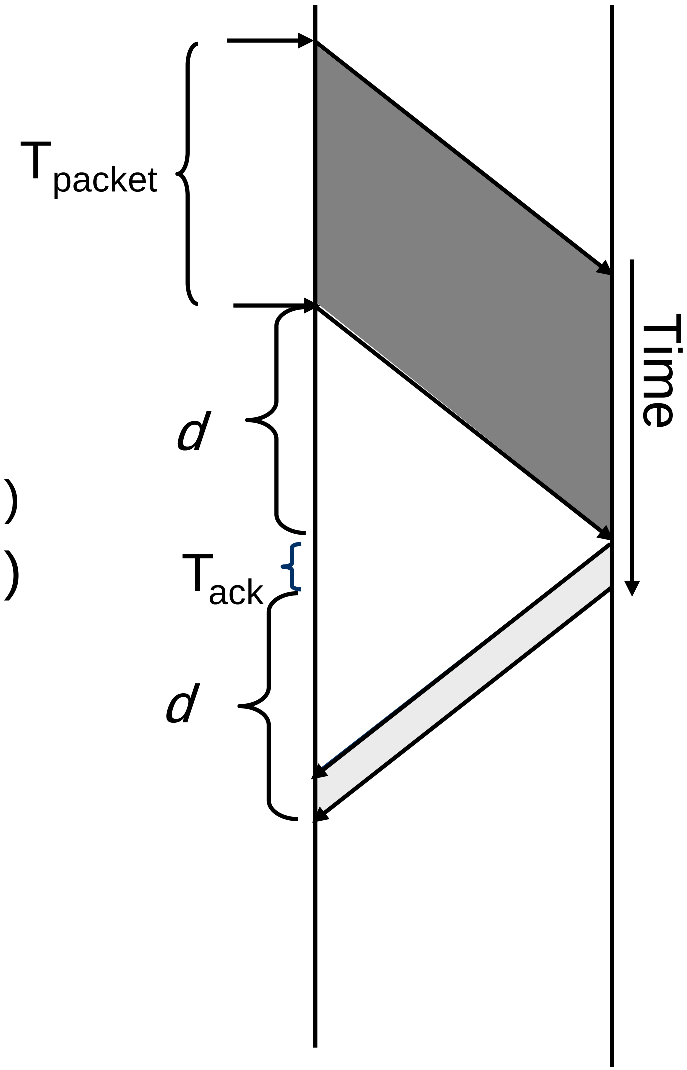
Alternating Bit Protocol

- Simple, but reliable protocol over noisy channels
- Uses 0 and 1 as sequence numbers, ARQ for retransmission
- Simple form of flow control (here combined with error control, other protocols separate these functions)



Alternating Bit Protocol – Efficiency

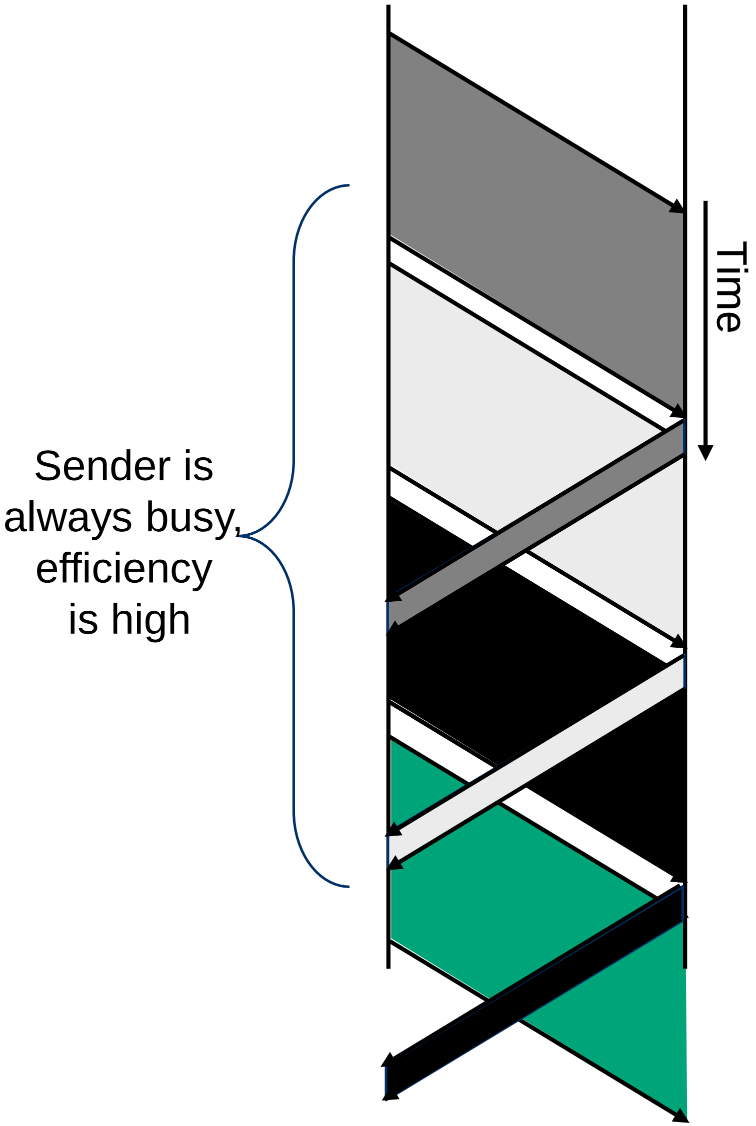
- Efficiency η:
  - Depends on delay and bandwidth
  - Defined as ratio of time during which sender sends new information
    - assuming error-free channel in simplest case (error-considerations make efficiency discussions difficult)
  - $\eta = T_{packet} \; / \; (T_{packet} + d + T_{ack} + d)$
- Efficiency of simple alternating bit protocol is low when delay is large compared to data rate
  - Bandwidth-delay product, i.e. data in transit, not used optimally



Improving Efficiency – Have More "Outstanding" Packets

- Inefficiency of alternating bit in large bandwidth-delay situations is owing to not exploiting "space" between packet and acknowledgement

- Always sending packets results in high efficiency
  - More packets are "outstanding" = sent, but not yet acknowledged
  - ➢ "Pipelining" of packets
- Not feasible with a single bit as sequence number
- ➢ Need larger sequence number space
  - Also needs full-duplex support



Handling Multiple Outstanding Packets

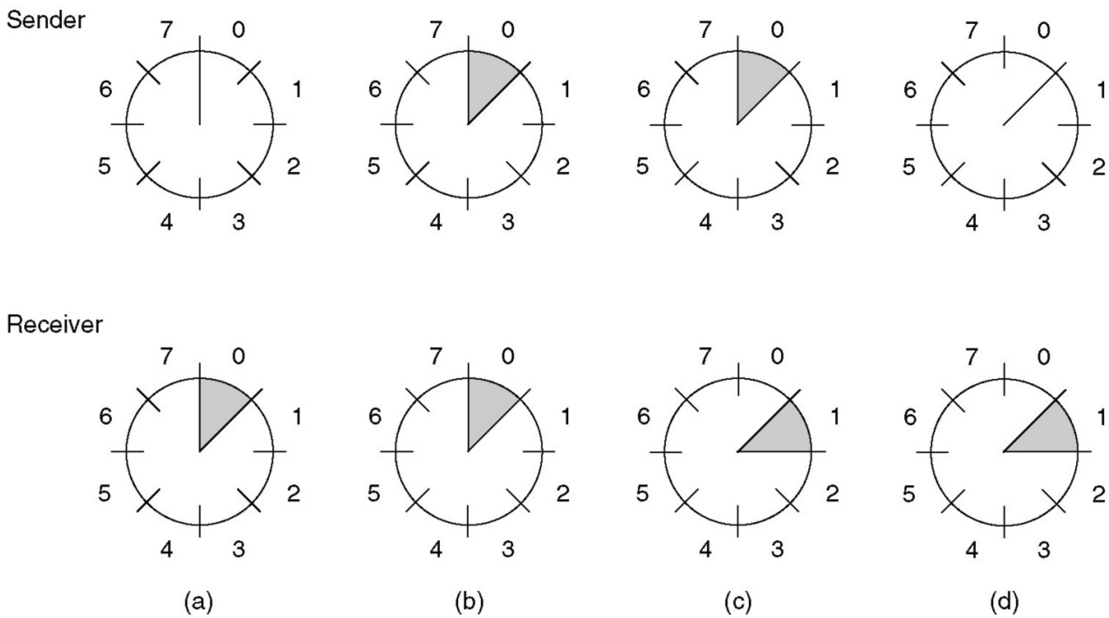
- Introduce a larger sequence number space
  - ➢ E.g., n bits or 2^n sequence numbers
- Not all of them may be allowed to be used simultaneously
  - Recall alternating bit case: 2 sequence numbers, but only 1 may be "in transit"

- Use *sliding windows* at both sender and receiver to handle these numbers
  - Sender: *sending window* – set of sequence numbers it is allowed to send at given time
  - Receiver: *receiving window* – set of sequence numbers it is allowed to accept at given time

➢ Window size corresponds to flow control
  - May be fixed in size or adapt dynamically over time

Simple Example: Sliding Window

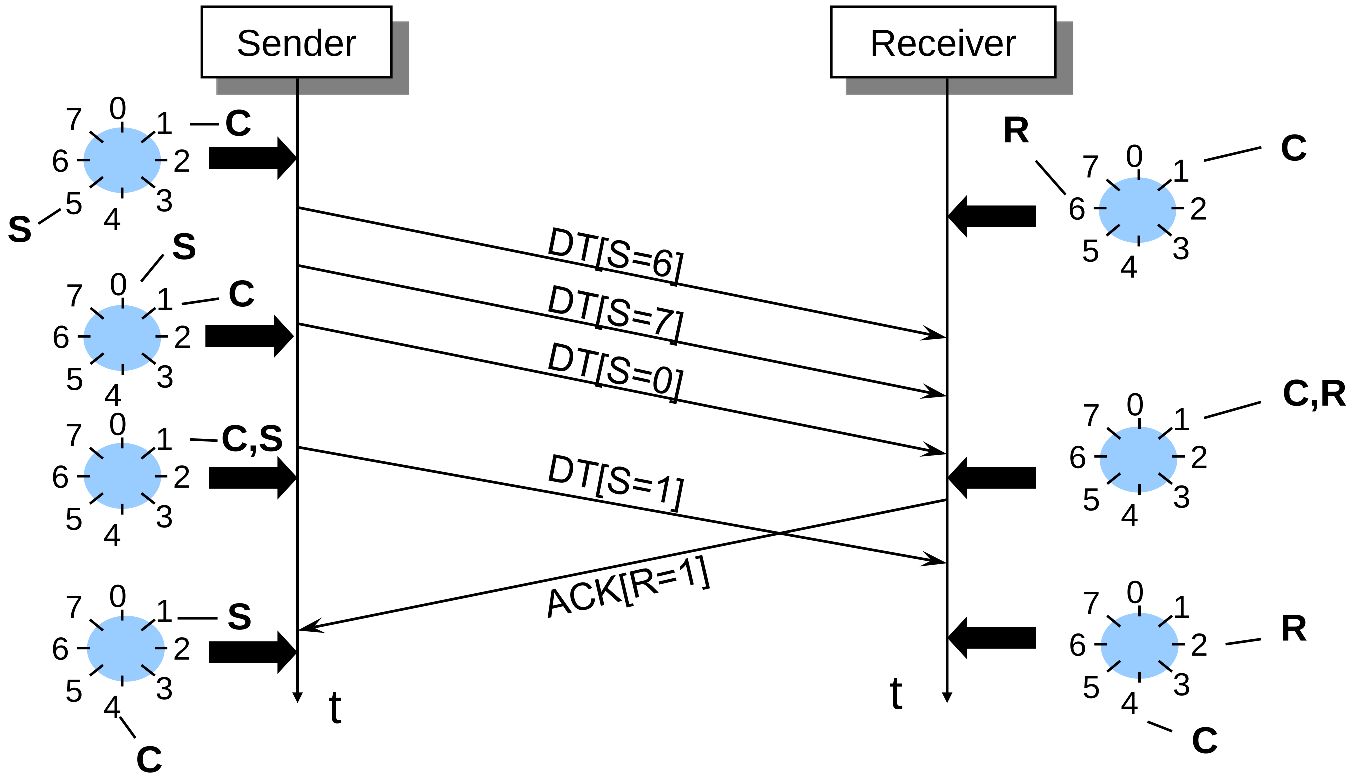
- Simple sliding window example for $n=3$, window size fixed to 1
- Sender tracks currently unacknowledged sequence numbers
  - If maximum number of unacknowledged frames is known, this is equivalent to sending window as defined on previous slide



a) Initially, before any frame is sent
b) After first frame is sent with sequence number 0
c) After first frame has been received
d) After first acknowledgement has arrived

Advanced Example: Sliding Window

- Sender credit (window size) = 4

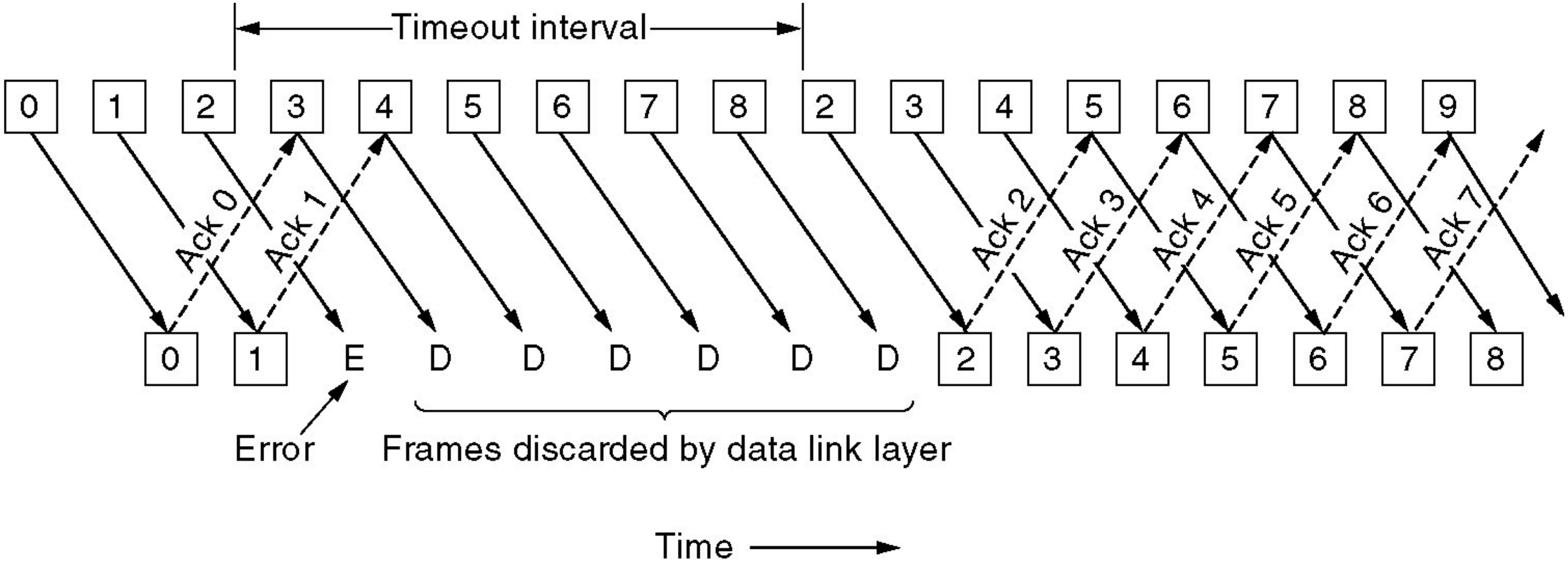


S: Sequence number (last sent packet)
R: Next expected sequence number = Acknowledges packets up to R-1
C: Upper window limit (current max. sequence number)

➢ Disadvantage: Coupling of flow control and error control

Transmission Errors and Receiver Window Size

- Assumption:
  - Link layer should deliver all frames correctly and in sequence
  - Sender is pipelining packets to increase efficiency
- What happens if packets are lost (discarded by CRC)?
- With receiver window size 1, all following packets are discarded as well!



Go-back-N

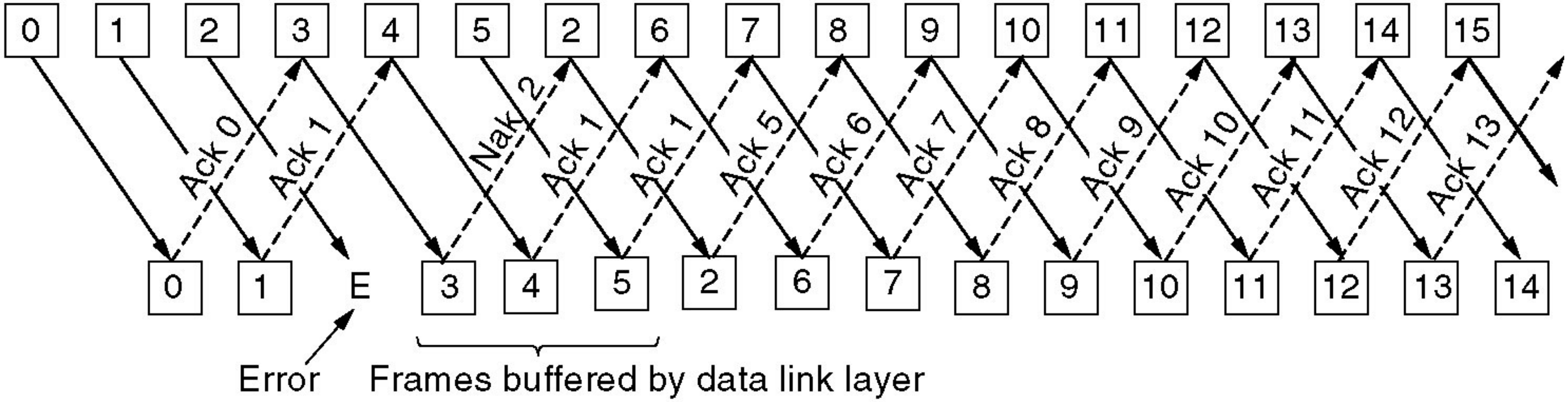
- With receiver window size 1, all frames following a lost frame cannot be handled by receiver
  - ➢ They are out of sequence
  - ➢ They cannot be acknowledged, only ACKs for the last correctly received packet can be sent

- Sender will timeout eventually
  - All frames sent in the meantime have to be repeated
  - ➢ Go-back-N (frames)

- Quite wasteful of transmission resources
- But saves resources at receiver

Selective Repeat

- Suppose we invest into a receiver that can buffer packets intermittently if some packets are missing
  - Corresponds to receiver window larger than 1
- Resulting behavior:



  - Receiver explicitly informs sender about missing packets using *Negative Acknowledgements* (NACKs)
  - Sender selectively repeats the missing frames
  - Once missing frames arrive, they are all passed to network layer
- More resources used at receiver, less overhead in case of error

Duplex Operation and Piggybacking

- So far, simplex operation at the (upper) service interface was assumed
  - Receiver only sent back acknowledgements, possibly using duplex operation of the lower layer service

- What happens when the upper service interface should support full-duplex operation?
  - Use two separate channels for each direction (SDMA)
    - Wasteful on bandwidth/resources
  - Interleave ACKs and data frames in a given direction (TDMA)
    - Better, but still some overhead
  - *Piggybacking*: Put ACKs from A to B into data frames from B to A (as part of B's header to A)
    - Minimal overhead
    - We'll see this principle again on layer 4 with TCP!

Conclusion

- Most problems in the link layer are due to errors:
  - Errors in synchronization require non-trivial framing functions
  - Errors in transmission require mechanisms to
    - Correct them so as to hide from higher layers
    - Detect them and repair them afterwards

- Flow control is often tightly integrated with error control in commonly used protocols
  - ➢ It *is* a separate function and can be implemented separately

- Connection setup/teardown still has to be addressed
  - ➢ Necessary to initialize a joint context for sender and receiver

Content (2)

8. Networked Computer & Internet
9. Host-to-Network I
10. **Host-to-Network II**
11. Host-to-Network III
12. Internetworking
13. Transport Layer