Telematics I

Chapter 4
Data Link Layer

- Link layer service and basic functions
- Framing
- Error control

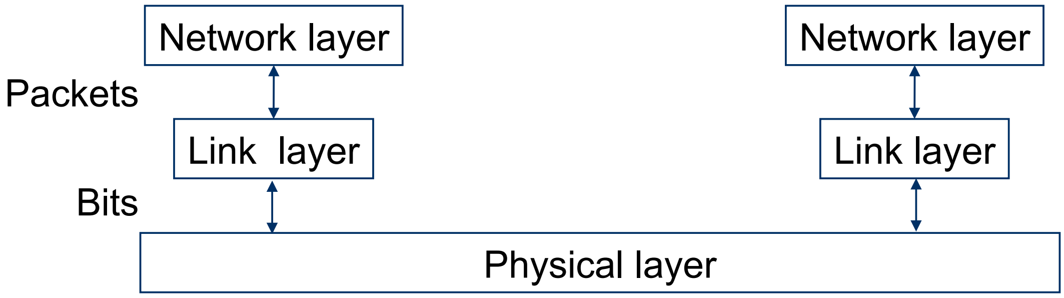
Goals

- Understand the main service provided by the link layer
  - Communication between two directly connected nodes
  - Framing of a physical bit stream into a structure of frames/packets
  - Error control: Detection and correction
  - Connection setup and release
  - Acknowledgement-based protocols
  - Flow control

- Some ideas about how to use extended finite state machines to specify communication protocols

The Link Layer's Service

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



Options for Link Layer Service

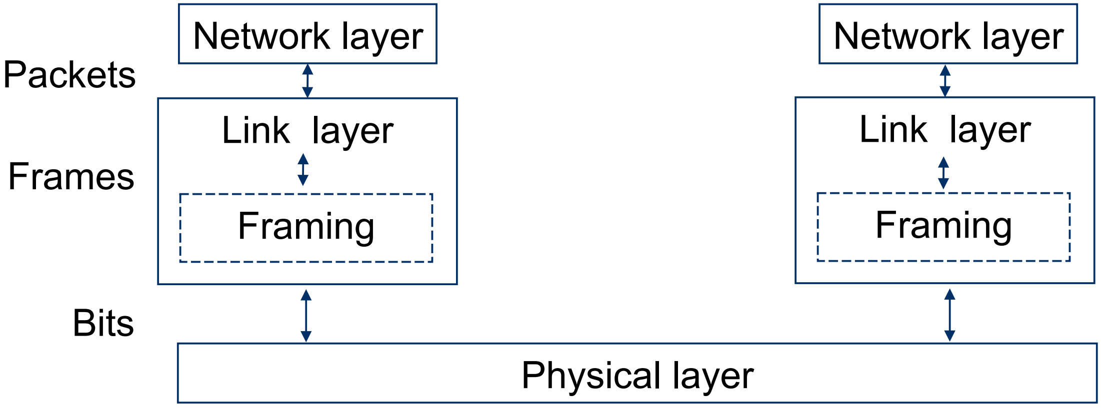
- Reliable (dependable) service – yes/no
  - Reliability has many facets
    - A delivered packet should have the same content as the transmitted packet
    - All packets have to be delivered eventually
    - Packets have to be delivered in order
  - ***Error control*** may be required
    - Forward error correction, or backward error correction with acknowledgements and retransmissions
- Connection-oriented – yes/no
  - Should a context be setup to/with the peer entity?
- Packet or bitstream abstraction
  - Usually in computer networks: packets
  - What about a maximal packet length?

Distinguish: Service Versus Implementation

- Note the difference between service and implementation
- One example:
  - Connection-less & reliable service required by the network layer
  - Link layer decides to use connections ***internally*** as a means to help with error control
- What about other combinations?

Basic Link Layer Functions – Framing

- How to turn a physical layer's bit stream abstraction 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, only a convention to talk about "frames" in the link layer context
- In addition: Fragmentation & reassembly if network layer packets are longer than link layer packets



Network layer

Network layer

Packets

Link layer

Link layer

Frames

Framing

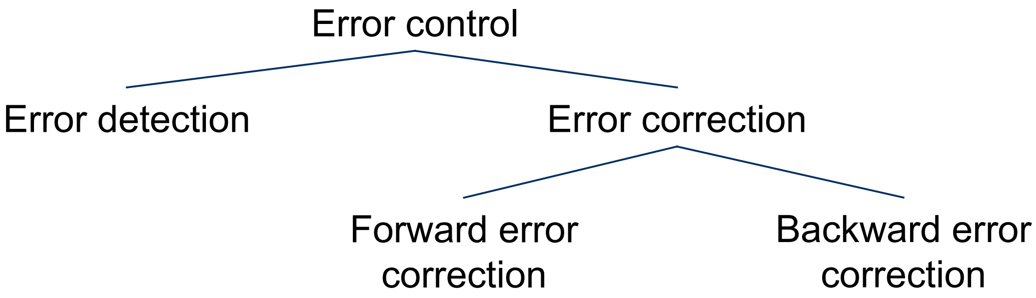
Framing

Bits

Physical layer

Basic Link Layer Functions – Error Control

- If desired by the network layer – usually is
- Usually build on top of frames
- Error detection – are there incorrect bits?
- Error correction – repair any mistakes that have happened?
  - Forward error correction – invest effort ***before*** error happened; try to hide it from higher layers
  - Backward error correction – invest effort ***after*** error happened; try to repair it



Basic Link Layer Functions – Connection Setup

- Connections (= common context) useful for many purposes
  - *Connections = common context*, e.g. application context
  - Error control – several error control schemes rely on a common context between sender and receiver
  - Don’t mix up connections with *circuits* = *(switched) common medium*
- Question: how to *set up and terminate* a connection? What state information is required?
  - Especially: if used on top of frames / packets?
  - A “virtual” connection, really, since there may be no end-to-end circuit switched
  - Example for a connection-oriented service on top of packet switching

- Problem reappears later in the transport layer again, with some additional complications – treated there!

Basic Link Layer Functions – Flow Control

- What happens with a fast sender and a slow receiver?
  - Sender will overrun buffers faster than the receiver can process the packets in that buffer
  - Lots of transmission effort is wasted in this case

Thirsty?
Drink!

- Necessary to control the amount of frames a link layer sends per unit time, adapt to receiver's capabilities

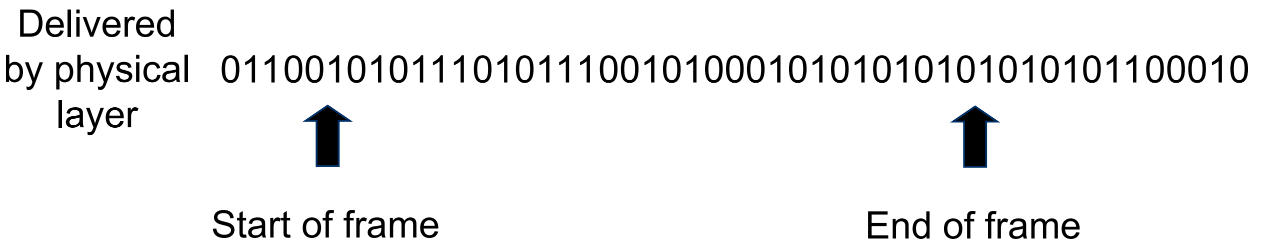
---

Content

- Link layer service and basic functions
- ***Framing***
- Error control

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 finishes?

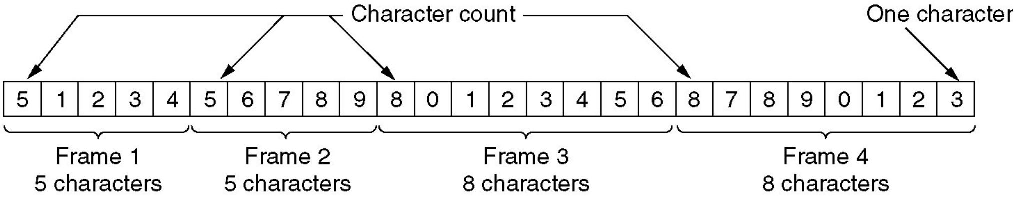


- Note: Physical layer might try to detect and deliver bits when the sender is not actually transmitting anything
  - Receiver still tries to get any information from the physical medium

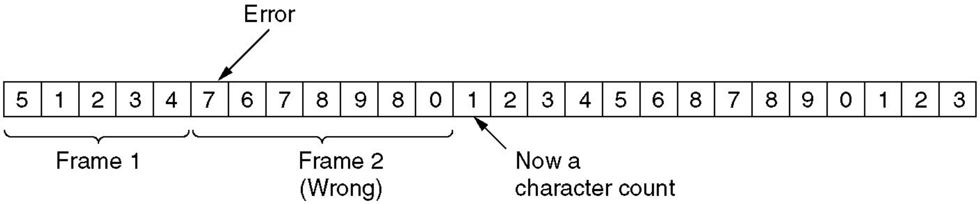
---

Framing by Character Counting

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

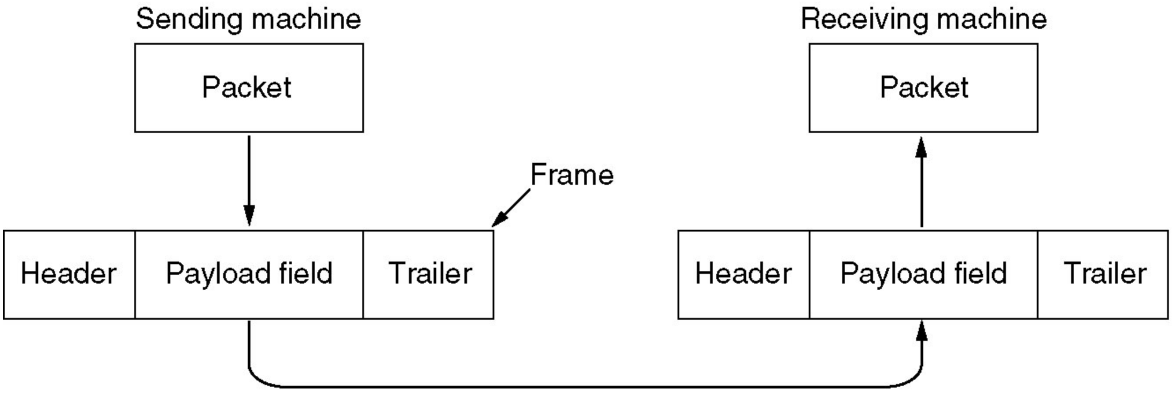


- Problem: What happens if the *count* information itself is damaged during transmission?
  - Receiver will loose frame synchronization and produce different sequence of frames than original one



Basic Technique: Put Control Data into a Header

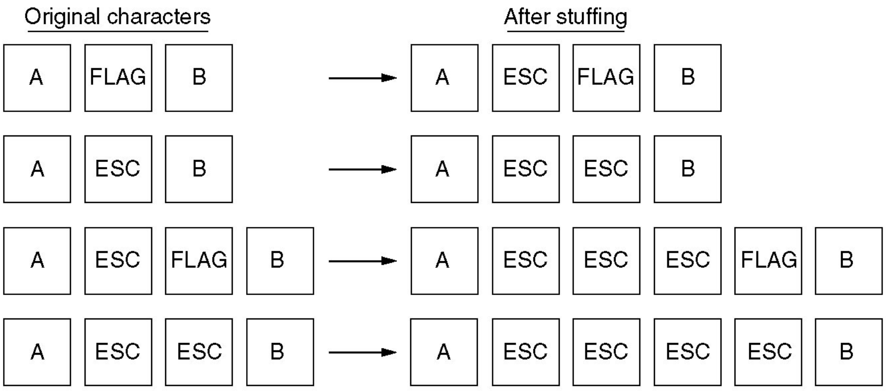
- Albeit “character count” is not a good framing technique, it illustrates an important technique: ***headers***
  - If sender has to communicate administrative or control data to receiver, it can be added to the ***payload,*** the actual packet content
  - Usually at the start of the packet; sometimes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



Framing by Flag Bytes/Byte stuffing

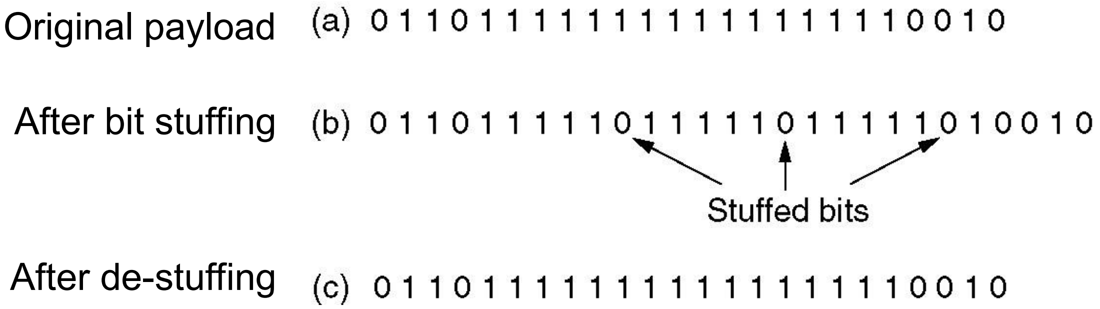
- Use dedicated ***flag bytes*** to demarcate start/stop of a frame

- What happens when 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 a bit pattern instead of a flag byte – often, 01111110
    - ■ Actually, it IS a flag byte
  - ❑ Use bit stuffing
    - ■ Whenever sender sends five 1’s in a row, it automatically adds a zero into the bit stream – except in the flag pattern
    - ■ Receiver throws away (“destuffs”) any 0 after five 1’s



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 the physical layer can express are used to express bit patterns
  - ❑ Example: Manchester encoding – only low/high and high/low is used
- ❑ When “violating” these encoding rules, data can be transmitted – e.g., the start and end of a frame
  - ❑ Example: Manchester – use high/high or low/low
    - ■ This drops the self-clocking feature of Manchester, but clock synchronization is sufficiently good to hold for a short while

- ❑ Powerful and simple scheme – used e.g. by Ethernet networks
  - ❑ But raises questions regarding bandwidth efficiency

Content

- Link layer service and basic functions
- Framing
- ***Error control***
  - ***Redundancy***
  - Hamming distance & error correction
  - Error detection – CRC
  - Backward error correction – Acknowledgement

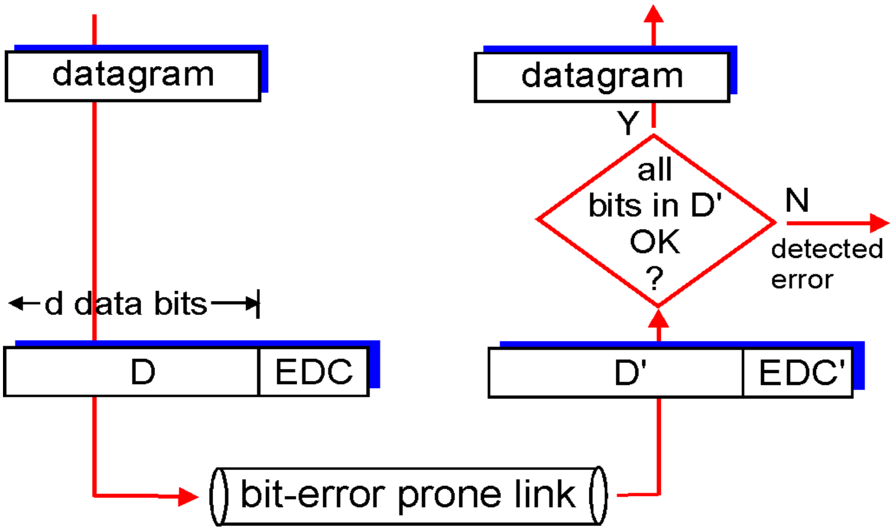
Error Control

- Two basic aspects:
  - ***Detect*** the presence of errors (incorrectly received bits) in a frame
  - ***Correct*** errors in frames
- Either one is possible without the other one
  - **Detect, but do not correct**: Simply drop a frame; pretend that it never has arrived at the receiver
    - Higher layers can take corrective measures, if they so desire
  - **Correct, but do not detect**: Try to correct as many errors as possible but do not care if there are some remaining errors present
    - Only feasible if application is not (too much) bothered by errors
    - Example: voice applications can tolerate some degree of bits errors without loosing too much voice quality
    - Justifiable, since even with detection the residual error probability is always > 0

Error Detection

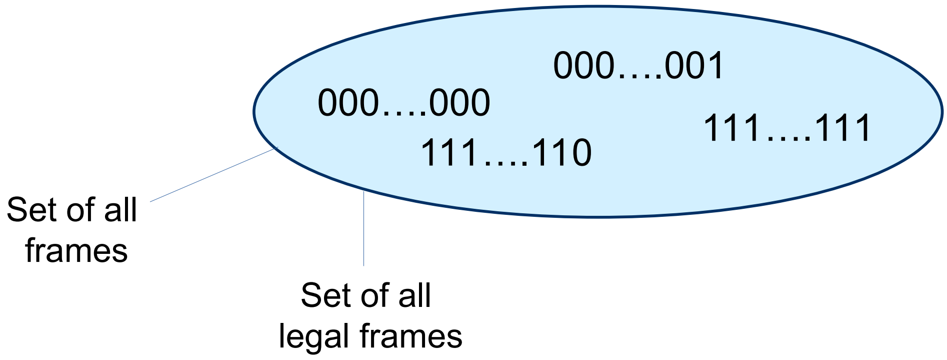
EDC = Error Detection and Correction bits (redundancy)
D = Data protected by error checking, may include header fields

- Error detection is not 100% reliable:
  - Protocol may miss some errors, but rarely
  - Larger EDC field yields better detection and correction



Error Control – Redundancy

- Any form of error control requires ***redundancy*** in the frames
- Without redundancy
  - A frame of length *m* can represent 2^m different frames
  - All of them are legal!
- How could a receiver possibly decide that one legal frame is not the one that had originally been transmitted?
  - Not possible!

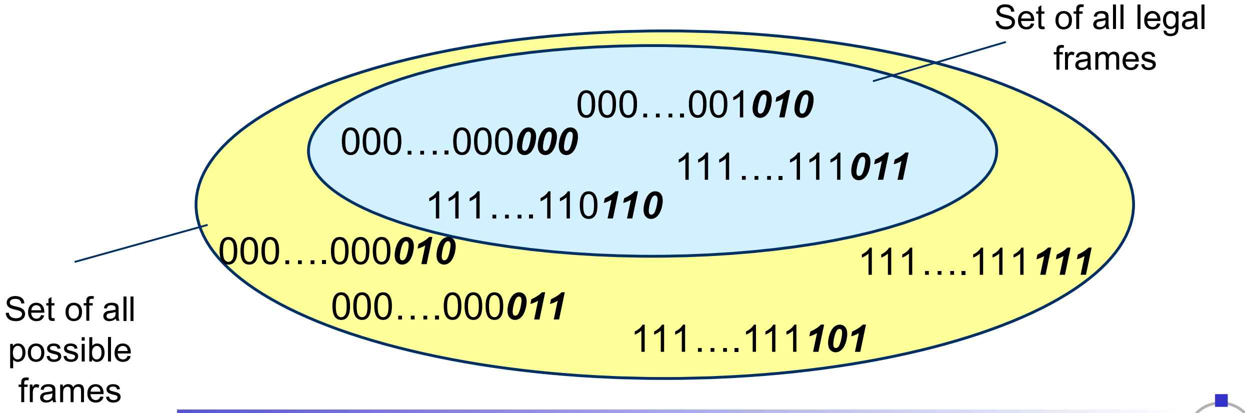


Error Control – Redundancy

- Core idea: Declare some of the possible messages illegal!
  - Still need to be able to express 2^m legal frames

  ! More than 2^m possible frames are required

  ! More than m bits are required in a frame
  - Use frames with $n > m$ total length
  - $r = m - n$ are the ***redundant bits*** (typically, as header or trailer)
- Having more *possible* than *legal* frames allows receiver to detect illegal frames

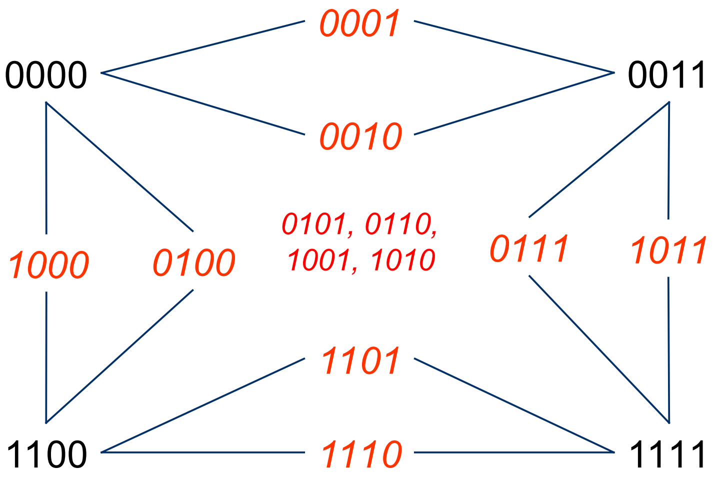


How Do Illegal Messages Help With Detecting Bit Errors?

- Transmitter only sends legal frame
- Physical medium/receiver might corrupt some bits
- Hope: A legal frame is only corrupted into an illegal message
  - But one legal frame is never turned into *another* legal frame
- Necessary to realize this hope:
  - Physical medium only alters up to a certain number of bits (by assumption) – say, k bits per frame
    - *This is only an assumption!*
    - How does it relate to the BER or the SNR?
  - Legal messages are sufficiently different so that it is not possible to change one legal frame into another by altering at most k bits

Altering Frames by Changing Bits

- Suppose the following frames are the only legal bit patterns: 0000, 0011, 1100, 1111



Lines connect frames that only differ in a single bit = that can be converted into each other by flipping one bit

Here: No single bit error can convert one legal frame into another one!

uvxy – legal frame *abcd* – illegal frame

Simple Redundancy Examples: Parity (1)

- A simple rule to construct 1 redundant bit (i.e., n = m + 1): ***Parity***
  - Odd parity: Add one bit, choose its value such that the number of 1’s in the entire message is odd
  - Even parity: Add one bit, choose its value such that the number of 1’s in the entire message is even

- Example:
  - Original message without redundancy: 01101011001
  - Odd parity: 01101011001***1***
  - Even parity: 01101011001***0***

Simple Redundancy Examples: Parity (2)

- Parity bit examples:
  - Send 1 0 1 1 0 0 0 in even parity
    - There are three 1’s in this
    - To make this even parity a 1 is added to the end (⇒ total four 1’s)
    - 1 0 1 1 0 0 0 1 is transmitted by the sending computer
  - Send 1 0 1 1 0 1 0 in even parity
    - There are four 1’s in this
    - To keep this even parity a 0 is added to the end
    - 1 0 1 1 0 1 0 0 is transmitted by the sending computer
  - The destination computer always expects an even number of 1’s:
    - If there is there is not an even number of 1’s arriving, then the frame has been corrupted

Simple Redundancy Examples: Parity (3)

- Parity bit problems:
  - Even and odd parity works well to detect single bit errors
  - However, it cannot detect all possible errors
  - For example, consider when transmission errors cause two bits to be changed:
    - If 1 0 0 1 1 0 1 0 is sent but two bits get changed during transmission
    - The destination computer receives 0 1 0 1 1 0 1 0 and does not realise that there were errors during transmission.

- To detect more errors (i.e. even number of bit errors), a *checksum* or a *cyclic redundancy check* is needed

Simple Redundancy Examples: Checksums (1)

- Checksums:
  - To compute a checksum, the sending computer treats the data as a sequence of binary integers and computes their sum.
  - Note that the Data Link Layer treats the data as a sequence of integers for the purposes of computing a checksum.
  - For example, to compute a checksum on the message “Hello World.”
  - Two characters are grouped together as a 16 bit number and added together to produce the checksum (adding potential carry-over at the end again like in computation of one-complement)

H	e	l	l	o		W	o	r	l	d	.
48	**65**	**6C**	**6C**	**6F**	**20**	**77**	**6F**	**72**	**6C**	**64**	**2E**

  - 4865 + 6C6C + 6F20 + 776F + 726C + 642E = 71FC
  - “Hello World.” is sent followed by 71FC

Simple Redundancy Examples: Checksums (2)

- Checksums:
  - Checksums are easy to calculate since they use simple addition and this can be done quickly by implementing it in hardware.
  - The disadvantage with checksums is that they cannot detect all common errors

Binary	Checksum value	Binary	Checksum value
0001	1	0011	3
0010	2	0000	0
0011	3	0001	1
0001	1	0011	3
Totals	7		7

Content

- Link layer service and basic functions
- Framing
- ***Error control***
  - Redundancy
  - ***Hamming distance & error correction***
  - Error detection – CRC
  - Backward error correction – Acknowledgement

Distance Between Frames

- In previous example: Two bit changes necessary to go from one legal frame to another
- Formally: ***Hamming distance***
  - Let x = $x_1,\dots, x_n$ and y = $y_1,\dots, y_n$ be frames
  - ***d(x,y) = number of 1 bits in x XOR y***
  - Intuitively: the number of bit positions where x and y are different

Example:

x=0011010111
y=0110100101
x XOR y=0101110010

d(x,y) = 5

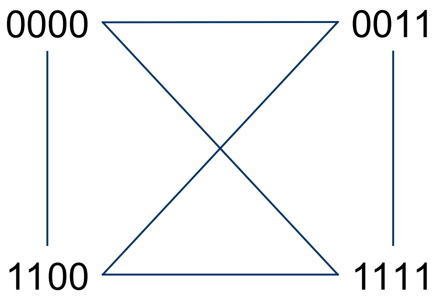
Hamming Distance of a Set of Frames

- The Hamming distance of a set of frames S:

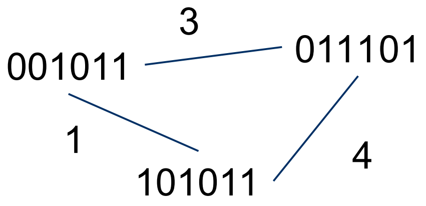
$$d(S) = \text{min}_{x,y \in S, x \neq y}\, d(x,y)$$

  - The smallest distance between any two frames in the set

Examples:



All distances are 2



One distance is 1!

Hamming Distance and Error Detection/Correction

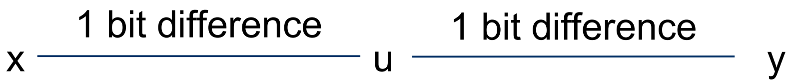
- What happens if d(S) = 0?
  - This is nonsense, by definition
- What happens if d(S) = 1?
  - There exist x,y ∈ S such that d(x,y) = 1; no other pair is closer

$$x \overset{\text{1 bit difference}}{\rule{4cm}{0.4pt}} y$$

  - A single bit error converts from one legal frame x to another legal frame y
  - Cannot detect or correct anything

Hamming Distance and Detection/Correction

- What happens if d(S) = 2?
  - There exist x,y ∈ S such that d(x,y) = 2; no other pair is closer
  - In particular: any u with d(x,u) = 1 is illegal,
    - As is any u with d(y,u)=1



  - I.e., errors which modify a single bit always lead to an illegal frame
    ! Can be detected!
  - Generalizes to all legal frames, because Hamming distance describes the “critical cases”
  - But not corrected – upon receiving u, no way to decide whether x or y had been sent (symmetry!)

Hamming Distance and Detection/Correction

- What happens if d(S) = 3?
  - There exist x, y ∈ S such that d(x,y) = 3; no other pair is closer
  - Every s with d(x,s) = 1 is illegal AND d(y,s) > 1!

  - Hence: the receipt of s could have the following causes:
    - Originally, x had been sent, but 1 bit error occurred
    - Originally, y had been sent, but 2 bit errors occurred
    - (Originally, some other frame had been sent, but at least 2 bit errors occurred)
  - Assuming that fewer errors have happened, a received frame s can be mapped to a frame x!
    - Hence, the error has been “corrected” – hopefully, correctly!

Generalization – Required Hamming Distances

- The examples above can be generalized
- To ***detect*** d bit errors, a Hamming distance of ***d+1*** in the set of legal frames is required
  - So that it is not possible to re-write a legal frame into another one using at most d bits
- To ***correct*** d bit errors, a Hamming distance of ***2d+1*** in the set of legal frames is required
  - So that all frames that are at most d bits away from a legal frame are illegal and are ***more*** than d bits away from any other legal frame

---

Frame Sets – Code Books, Codes

- A terminology aspect:
  - The set of legal frames $S \subseteq \{0,1\}^n$ is also called a ***code book*** or simply a ***code***
  - The ***rate R*** of a code S is defined as:
    - Rate characterizes the efficiency

$$R_S = \frac{\log |S|}{n}$$

  - The ***distance*** δ of a code S is defined as:
    - Distance characterizes error correction/detection capabilities

$$\delta_S = \frac{d(S)}{n}$$

- A good code should have large distance and large rate – but arbitrary combinations are not possible
  - For details: Information theory, Claude Shannon

How to Construct Error Correcting Codes

- Constructing good codes (e.g., highest rate at given error correction needs) is difficult
- Simple example: use several parity bits
    - Distribute the parity bits over the entire codeword to protect against burst errors

---

Content

- Link layer service and basic functions
- Framing
- ***Error control***
    - Redundancy
    - Hamming distance & error correction
    - ***Error detection – CRC***
    - Backward error correction – Acknowledgement

How to Construct Error Detecting Codes – CRC

- Efficient error detection: ***Cyclic Redundancy Check (CRC)***
- Gives rules how to compute redundancy bits and how to decide whether a received bit pattern is correct
  - Very high detection probability with few redundancy bits
  - Can be efficiently implemented in hardware
- Basic operation based on polynomial arithmetic
  - Bit string is interpreted as representing a polynomial
  - Coefficients 0 and 1 are possible, interpreted modulo 2

Modulo 2 Arithmetic

- With 0 and 1 as the only possible numbers (bits!), normal arithmetic is not applicable
- Instead: look at modulo 2 arithmetic
- Rules:
  - Addition modulo 2 Subtraction modulo 2 Multiplication modulo 2

A	B	A + B
0	0	0
0	1	1
1	0	1
1	1	0

A	B	A - B
0	0	0
0	1	1
1	0	1
1	1	0

A	B	A × B
0	0	0
0	1	0
1	0	0
1	1	1

  - Example: 0110111011
    + 1101010110 = 1011101101

Modulo 2 Division

- Division of two numbers is done just like normal division:
  - Subtract the denominator (the bottom number) from the leading parts of the enumerator (the top number)
  - Proceed along the enumerator until its end is reached
  - Remember that we are using modulo 2 subtraction.

- `1101010110 / 1001 = 1100110`

```
1001
 1000
 1001
  001101
    1001
     1001
     1001
         0
```

Modulo 2 Division With Remainder

- After division, a remainder may result

- `1101010101 / 1001 = 1100110` remainder `11`

```
1001
 1000
 1001
  001101
    1001
     1000
     1001
      0011
```

Polynomials Over Modulo 2 Arithmetic

- Define polynomials over modulo 2 arithmetic
  - $p(x) = a_n x^n + \ldots + a_1 x^1 + a_0$
  - Coefficients a_i and $x \in \{0,1\}$
  - Multiplication and addition is defined modulo 2
- Addition, subtraction, multiplication and division of polynomials is defined in the usual way!

Bit Strings and Polynomials Modulo 2

- Idea: Conceive of a string of bits as a representation of the coefficients of a polynomial
- Bit string: $b_n b_{n-1} \ldots b_1 b_0$

  Polynomial: $b_n x^n + \ldots + b_1 x^1 + b_0$
  - A bit string of (n+1) bits corresponds to a polynomial of degree n
- Operations on bit strings correspond to operations on polynomials and vice versa
  - Example: “Append *k* zeros” ⇔ “multiply by x^k”
- This isomorphism allows us ***to divide bit strings***!

Use Polynomials to Compute Redundancy Bits – CRC

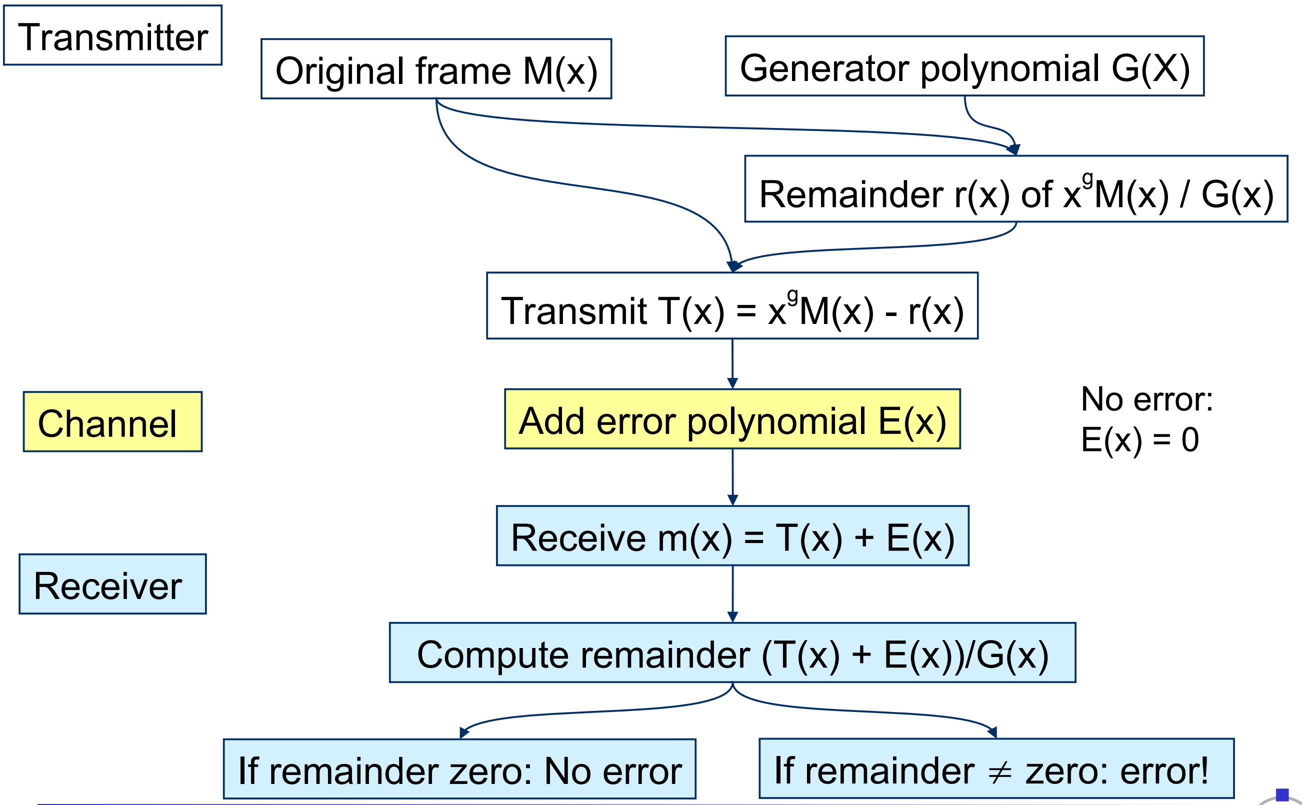
- Define a ***generator polynomial*** *G(x)* of degree *g*
  - Known to both sender and receiver
  - We will use *g* redundancy bits in the end
- Given: message/frame M, represented by polynomial M(x)
- Transmitter
  - Compute remainder r(x) of division $x^g M(x) / G(x)$
    - Note: Remainder after division is of degree < g, fitting into g bits!
  - Transmit $T(x) = x^g M(x) – r(x)$
    - Note: $x^g M(x) – r(x)$ is divisible without remainder by G(x)
- Receiver
  - Receive m(x)
  - Compute remainder of division of m(x) by G(x)

CRC Transmission and Reception

- What happens in the channel after transmitting T(x)?
  - No errors: T(x) arrives correctly at the receiver
  - Bit errors occur: T(x) is modified by flipping some bits
    - Equivalent to modifying some coefficients of the polynomial
    - Equivalent to adding an ***error polynomial*** E(x)
    - At the receiver, T(x) + E(x) arrives

- At the receiver
  - Receive m(x)
  - Compute remainder of division of m(x) by G(x)
  - No errors: m(x) = T(x). Remainder is zero!
  - Bit errors: $m(x)/G(x) = (T(x) + E(x)) / G(x) = \underbrace{T(x)/G(x)}_{\text{no remainder}} + \underbrace{E(x)/G(x)}_{\text{remainder usually not zero!}}$

CRC – Overview



Transmitter

Original frame $M(x)$

Generator polynomial $G(X)$

Remainder $r(x)$ of $x^gM(x) / G(x)$

Transmit $T(x) = x^gM(x) - r(x)$

Channel

Add error polynomial $E(x)$

No error: $E(x) = 0$

Receiver

Receive $m(x) = T(x) + E(x)$

Compute remainder $(T(x) + E(x))/G(x)$

If remainder zero: No error

If remainder $\neq$ zero: error!

Choice of G(x) Determines CRC Properties

- When is remainder of $E(x) / G(x) \neq 0$?
  - If $G(x)$ divides $E(x)$ without remainder, an error slips through!
- Single bit error: $E(x) = x^i$ for error at position i
  - $G(x)$ needs two or more terms to ensure that $E(x)$ is not a multiple of it

Two bit error: $E(x) = x^i + x^j = x^j (x^{i-j} +1)$ for some $i>j$
  - x must not divide $G(x)$
  - $G(x)$ must not divide $(x^k + 1)$ for all k up to, e.g., maximum frame length
- Odd number of errors: $E(x)$ has an odd number of terms
  - $E(x)$ will NOT have $(x+1)$ as a factor
  - Make $(x+1)$ a factor of $G(x)$ so that it cannot divide $E(x)$
- Using r check bits, all burst errors of length $< r$ can be detected (as well as “most” burst errors of length $\geq r$)

Commonly Used CRC Generator Polynomials

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

- In practice, residual errors after CRC check are ignored
  - But they may still happen!
  - In particular, when bit errors are not independent, but bursty

---

Content

- Link layer service and basic functions
- Framing
- ***Error control***
  - Redundancy
  - Hamming distance & error correction
  - Error detection – CRC
  - ***Backward error correction – Acknowledgement***

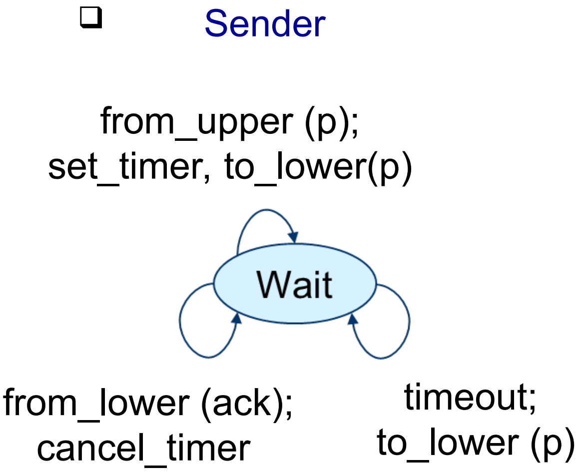
How to Handle Detected Errors?

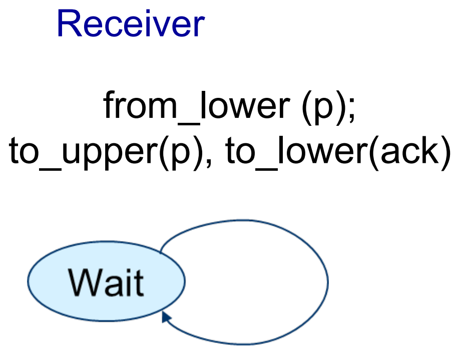
- Suppose the receiver detects an error
- Clearly, the received frame cannot be delivered to higher layers/application

  ! Have to ***repair*** the error somehow
- Two principle approaches:
  - **Forward:** sender sends redundant information so that receiver can correct “a couple of” errors (requires advanced coding techniques not covered in this course)
  - **Backward:** sender sends redundant information so that receiver can detect errors with high probability and upon detection of an error, packets are retransmitted
- Backward correction protocols are generally known under the name **Automatic Repeat Request (ARQ)**, denoting three main variants:
  - Send and wait
  - Go-Back-N
  - Selective reject (selective retransmission)

A Simple, Simplex, Acknowledgement-Based Protocol

- Acknowledge to sender the receipt of a packet
  - Sender waits for acknowledgement for a certain time
  - If not received in time, packet is retransmitted
- First solution attempt:





Note: to_lower, from_lower take care of CRC (FEC, if desired)

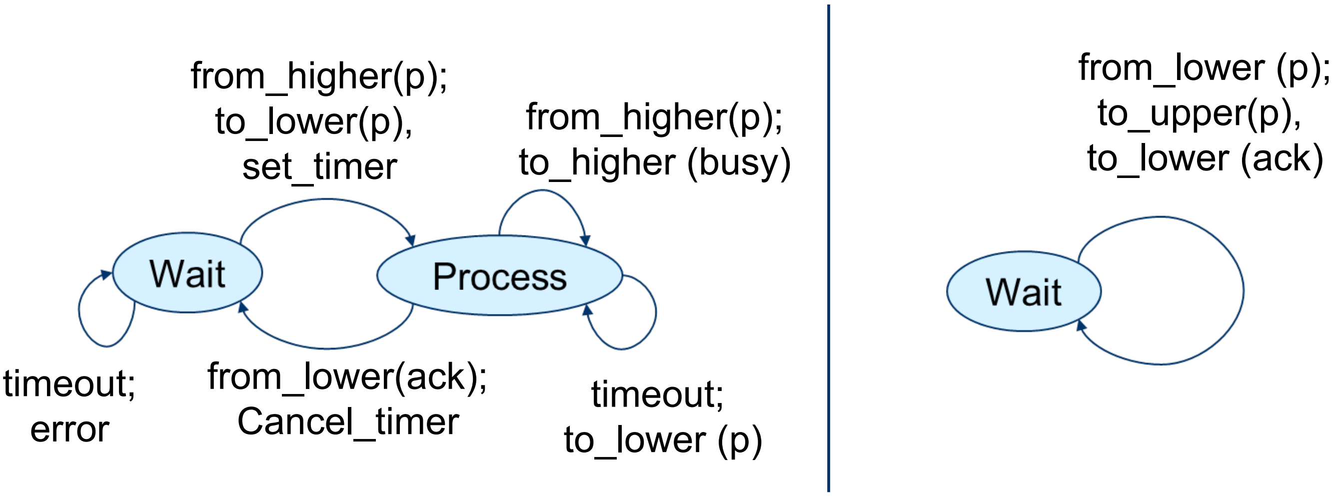
Protocol Analysis

- This protocol is nice and simple, but flawed in multiple ways
  - What happens when the higher layer sends packets faster than the acknowledgements come in (and when one is missing?)
  - What happens if acknowledgements are lost?

- Need some repairs here…

Acknowledgement-Based Protocol, Second Trial

- Cure one problem: Concentrate on one packet, only accept the next packet from higher layer when previous one has been fully processed
- First solution attempt:
  - Sender Receiver



from_higher(p);
to_lower(p),
set_timer

from_higher(p);
to_higher (busy)

Wait

Process

timeout;
error

from_lower(ack);
Cancel_timer

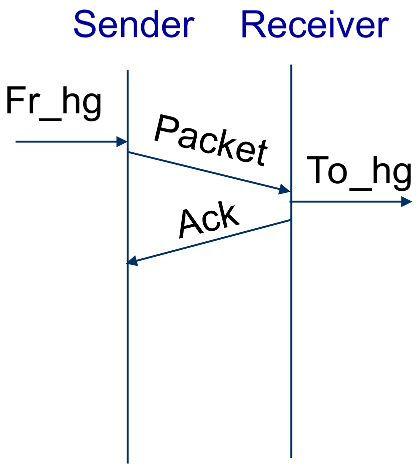
timeout;
to_lower (p)

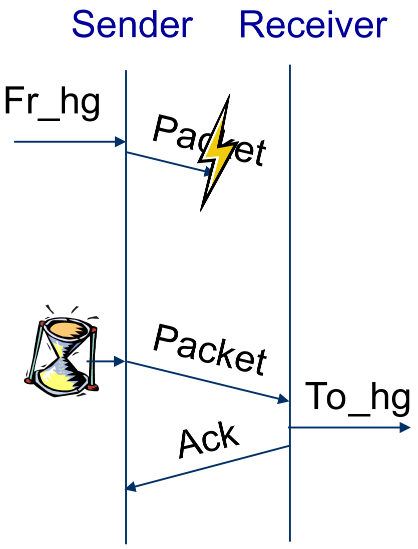
from_lower (p);
to_upper(p),
to_lower (ack)

Wait

Does Second Version Work Correctly?

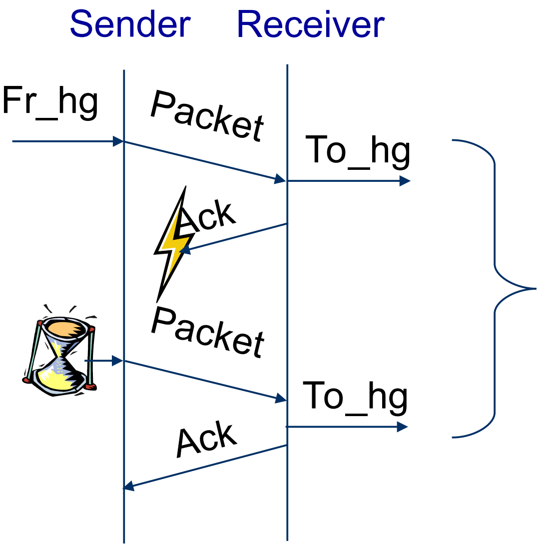
- It holds back the transmitter until packets are processed
  - It implements ***flow control***!
- Does it ensure that all packets arrive, in correct order?





Does Second Version Work Correctly?

- Simple cases seem ok
- What if an *acknowledgement is lost?*



Same packet is delivered twice to the receiver's higher layer!

Summary of Second Version: Send and Wait

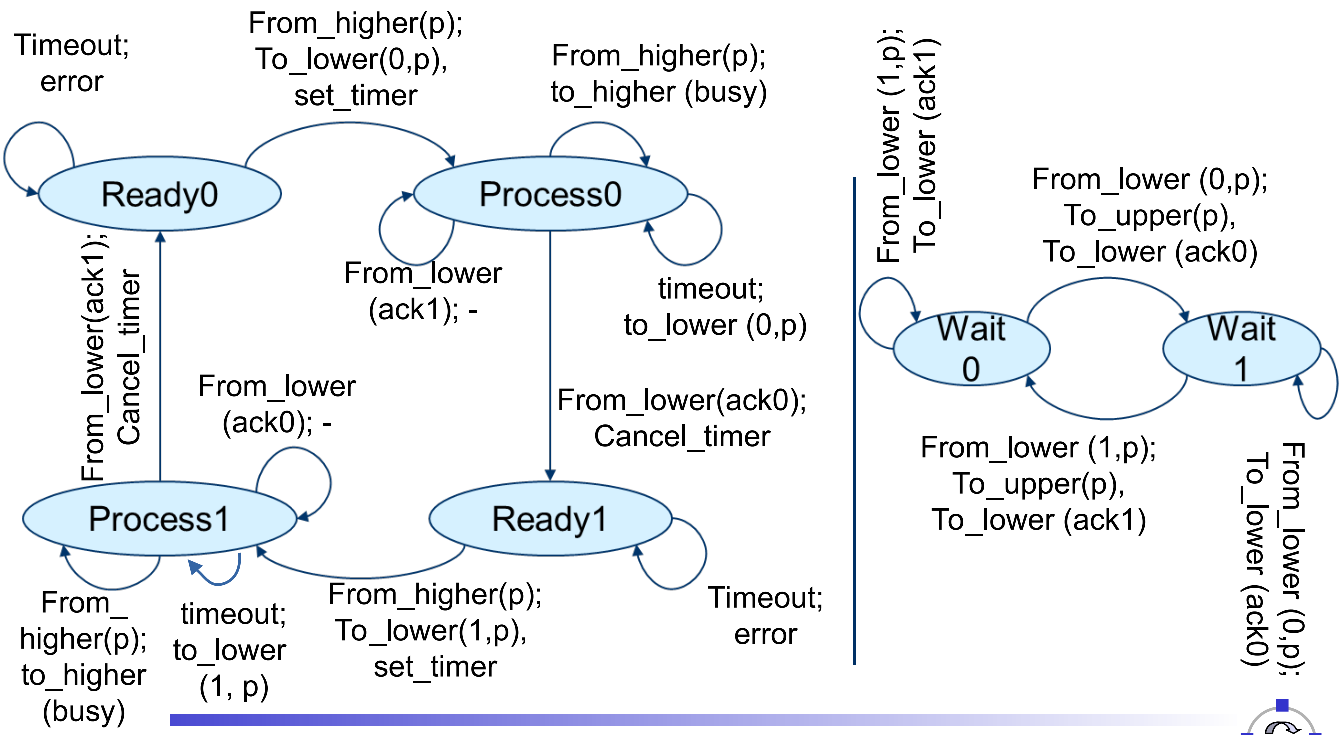
- Sender transmits one single packet:
  - Sender sets a timeout
  - Sender waits for acknowledgement (ACK)
  - If no ACK is received within timeout, the sender retransmits the packet
- If a received packet is damaged, the receiver simply discards it
- Often, this scheme is also referred to as *“Stop-and-Wait”* as the sender stops transmitting after each packet
- If the ACK packet is damaged,the sender will not recognize it:
  - Sender will also retransmit the packet
  - Receiver gets two copies of packet

Overcoming the Problem of Send and Wait

- Sender cannot distinguish between a lost packet and a lost acknowledgement
  ! Has to re-send the packet
- Receiver cannot distinguish between a new packet and a redundant copy of an 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: a 0 or 1 !
- Needed in packet & acknowledgement
  - In Ack, convention: send the sequence number of the last correctly received packet back
    - Also possible: send sequence number of next expected packet

Acknowledgements & Sequence Numbers – 3rd Version



Sender

Receiver

Timeout;
error

From_higher(p);
To_lower(0,p),
set_timer

From_higher(p);
to_higher (busy)

Ready0

Process0

From_lower
(ack1); -

timeout;
to_lower (0,p)

From_lower(ack1);
Cancel_timer

From_lower
(ack0); -

From_lower(ack0);
Cancel_timer

Process1

Ready1

From_
higher(p);
to_higher
(busy)

timeout;
to_lower
(1, p)

From_higher(p);
To_lower(1,p),
set_timer

Timeout;
error

From_lower (1,p);
To_lower (ack1)

From_lower (0,p);
To_upper(p),
To_lower (ack0)

Wait
0

Wait
1

From_lower (1,p);
To_upper(p),
To_lower (ack1)

From_lower (0,p);
To_lower (ack0)

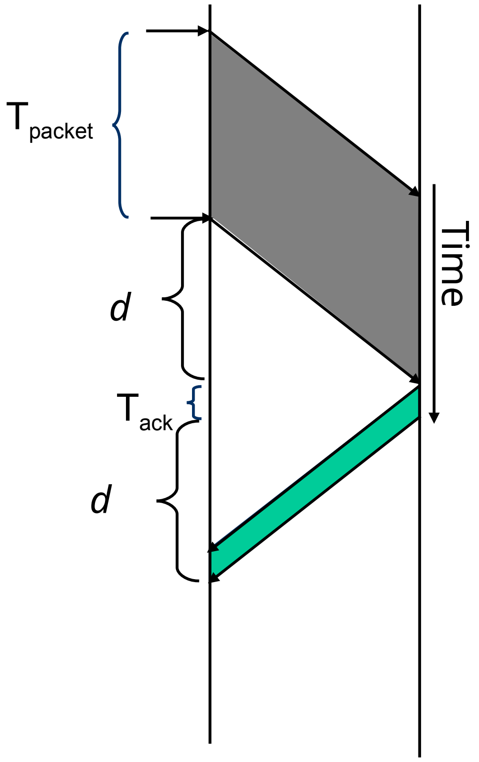
Assessment of 3rd Version – Alternating Bit Protocol

- This 3rd version is a correct implementation of a reliable protocol over a noisy channel
  - Name: ***Alternating bit protocol***
  - Class of protocols where sender waits for a positive confirmation: ***Automatic Repeat reQuest (ARQ) protocols***
  - It also implements a simple form of flow control

- Note the dual task of the acknowledgement packet
  - It **confirms** to the sender that the receiver has obtained a certain packet
  - It is also the **permit** to send the next packet, stating that the receiver has the capacity to handle it
  - *These two functions can be and are separate in other protocols!*

Alternating Bit Protocol – Efficiency

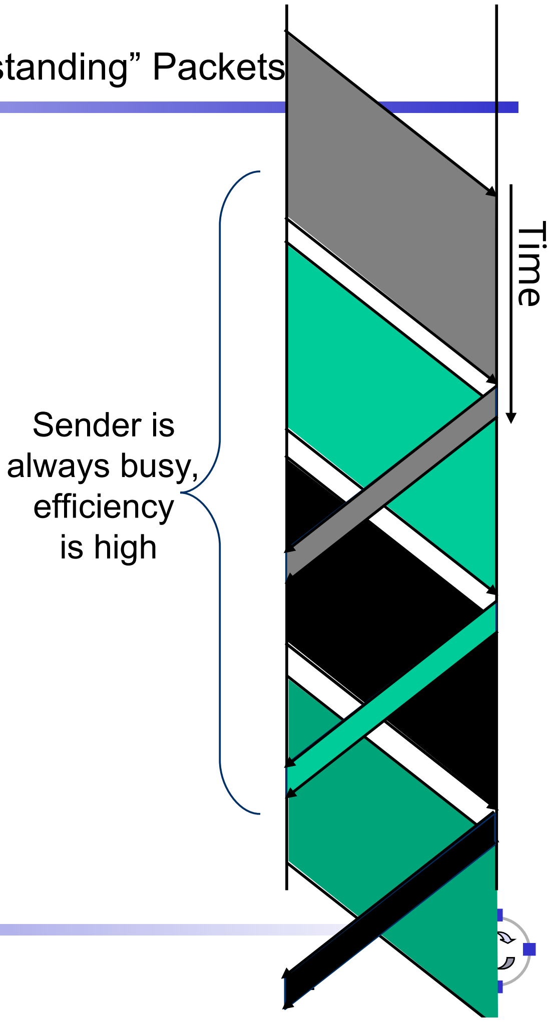
- Efficiency η: depends on circumstances
  - Defined as the ratio of time during which the sender sends new information (assuming an error-free channel in the simplest case; error-considerations make efficiency discussions difficult)
  - $\eta = T_{packet} / (T_{packet} + d + T_{ack} + d)$
- Efficiency of simple alternating bit protocol is low when delay is large compared to data rate
  - Recall bandwidth-delay product
  - This will be further discussed in a performance comparison later on...



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
- But not feasible with a single bit as sequence number

  ! Need larger sequence number space!
  - It also needs – ideally – some full-duplex support
  - How to live without full-duplex?



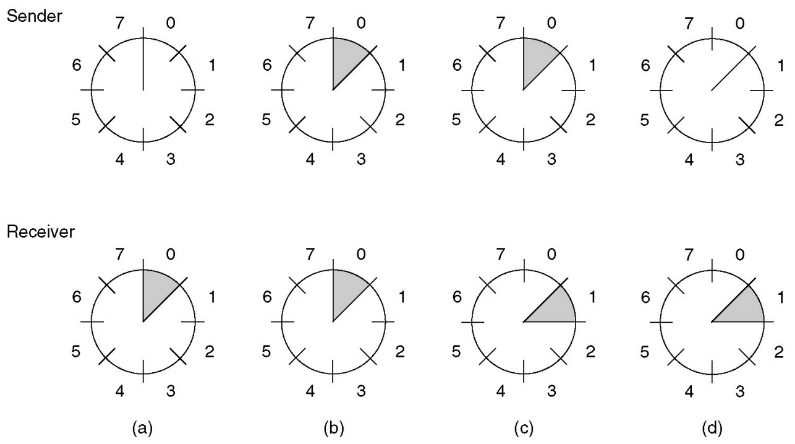
Sender is always busy, efficiency is high

Sliding Windows to Handle Multiple Outstanding Packets

- Introduce a larger sequence number space
  - Say, 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
  - May be fixed in size or adapt dynamically over time
  - Window size corresponds to flow control

Sliding Window – Simple Example

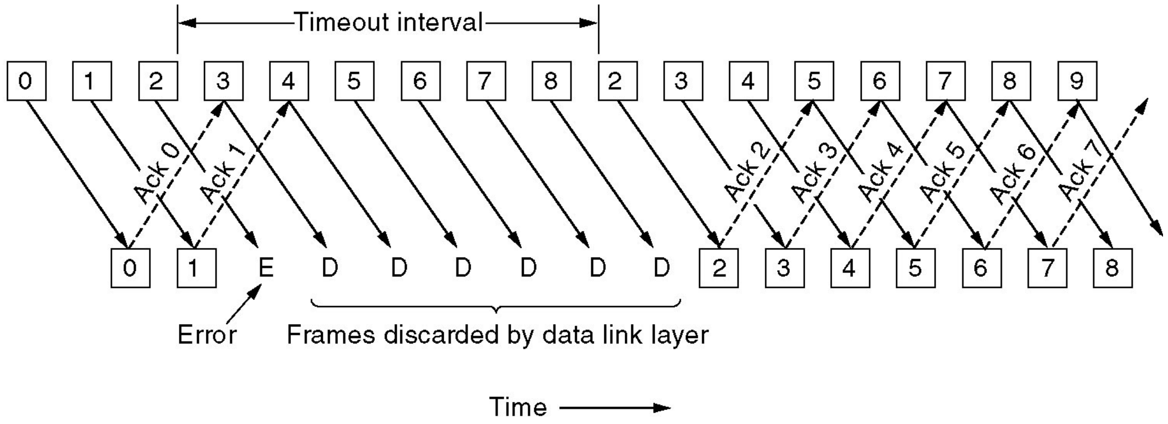
- A simple sliding window example for n=3, window size fixed to 1
- Sender here represents the currently unacknowledged sequence numbers
  - If maximum number of unacknowledged frames is known, this is equivalent to sending window as defined on previous slide



a. Initially, before any frame is sent
b. After first frame is sent with seq. num 0
c. After first frame has been received
d. After first acknowledgement has arrived

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!



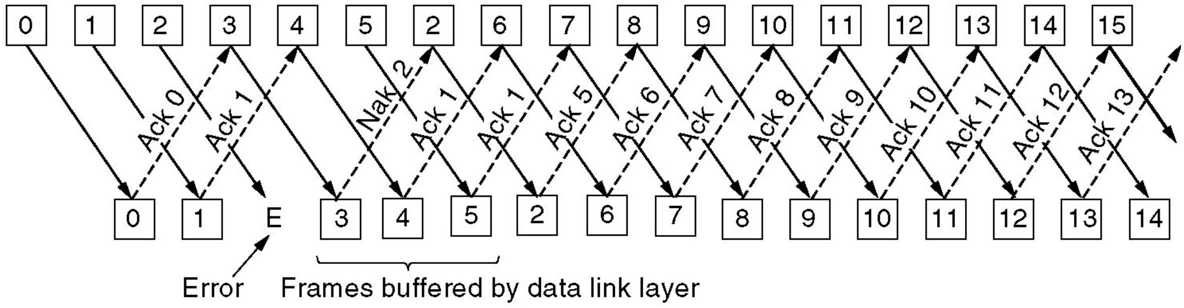
Go-Back-N

- 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
  - Since all frames sent in the meantime, they have to be repeated

    ! Go-back N (frames)!
  - Also called *Sliding Window ARQ*
- Assessment
  - Quite wasteful of transmission resources
  - But saves overhead at the receiver

Selective Reject (Selective Repeat)

- Suppose we invest a bit 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***
  - Sender selectively repeats the missing frames
  - Once missing frames arrive, they are all passed to the network layer

---

Duplex Operation and Piggybacking

- So far, simplex operation at the (upper) service interface was assumed
  - The 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?
  - One option: Use two separate channels for each direction – wasteful
  - Better: Interleave acknowledgement and data frames in a given direction
  - Best (and usual): Put the acknowledgement information for direction A! B into the data frames for B ! A
    - As part of B’s header – ***piggyback*** it

Performance: Error-Free Send and Wait (1)

- In order to assess the performance differences of the different protocols, let us compute the time for sending one packet and receiving and processing the respective acknowledgement:
  - $T = T_{frame} + T_{prop} + T_{proc} + T_{ack} + T_{prop} + T_{proc}$

  - T_{frame} = time to transmit frame
  - T_{prop} = propagation time
  - T_{proc} = processing time at station
  - T_{ack} = time to transmit ack

- Assume T_{proc} and T_{ack} relatively small:
  - $T \approx T_{frame} + 2T_{prop}$

(Acknowledgement: figures in performance discussion according to a prior edition of [Sta04])

Send and Wait Link Utilization

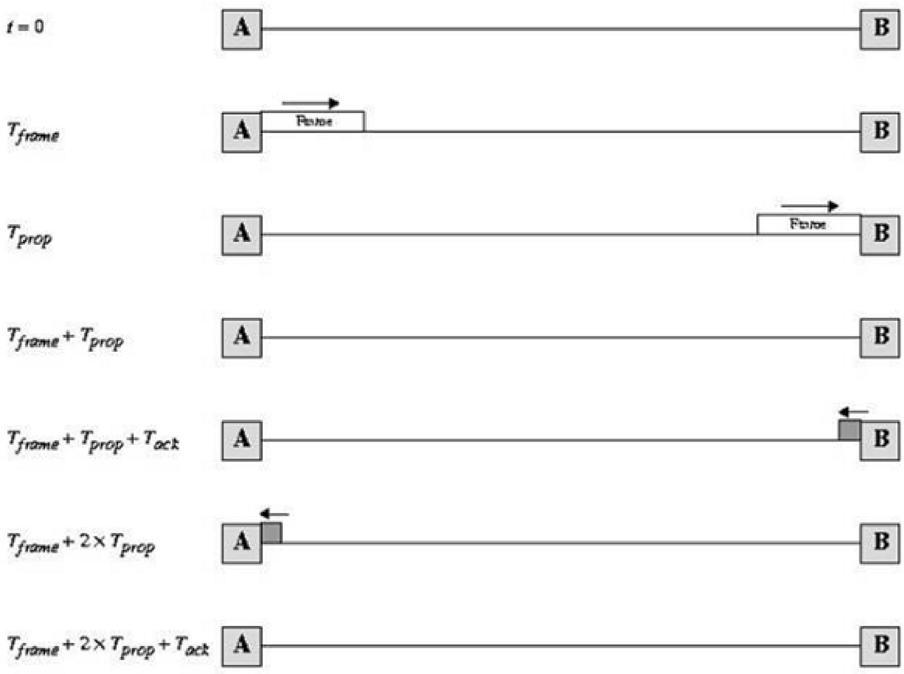


Figure 11.4 Stop-and-Wait Link Utilization

Performance: Error-Free Send and Wait (2)

Throughput = $1/T = 1/(T_{frame} + 2T_{prop})$ frames/sec

Normalize by link data rate: $1/T_{frame}$ frames/sec

$$S = \frac{1/(T_{frame} + 2T_{prop})}{1/T_{frame}} = \frac{T_{frame}}{T_{frame} + 2T_{prop}} = \frac{1}{1 + 2a}$$

where $a = T_{prop} / T_{frame}$

Performance: Send-and-Wait ARQ with Errors

P = probability a single frame is in error

$$N_x = \frac{1}{1 - P}$$

= average number of times each frame must be transmitted due to errors

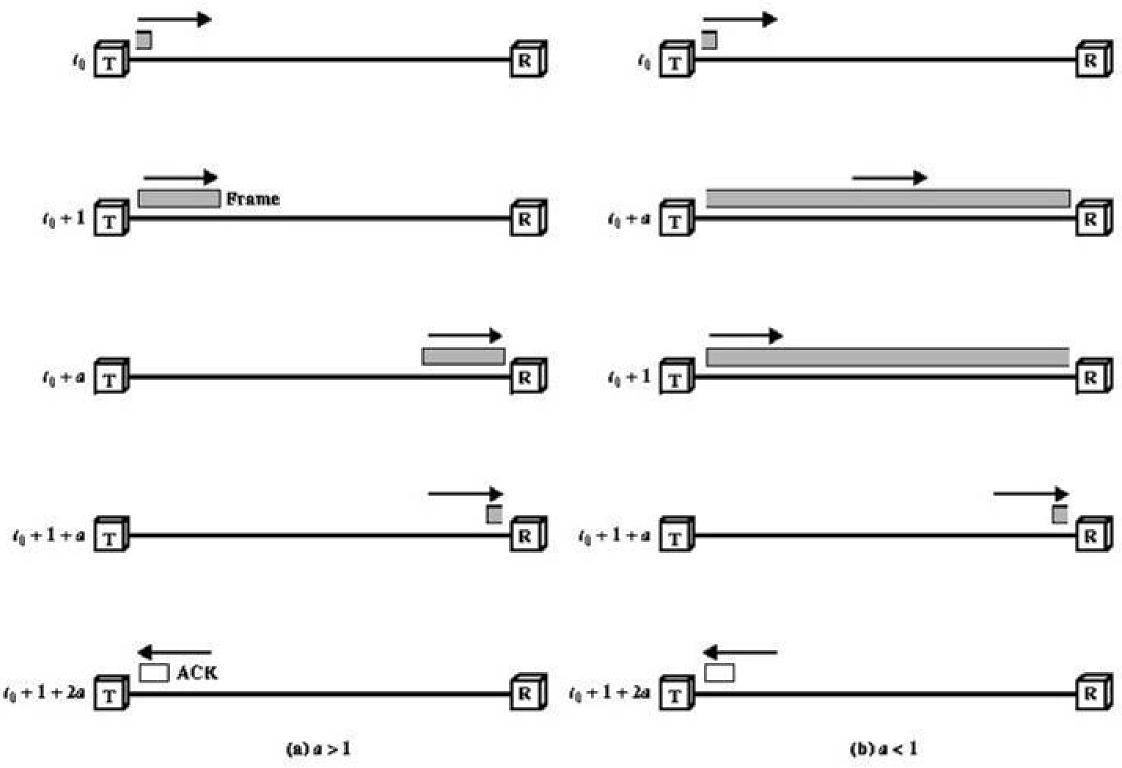
$$S = \frac{1}{N_x (1 + 2a)} = \frac{1 - P}{(1 + 2a)}$$

The Parameter a

$$a = \frac{\text{propagation time}}{\text{transmission time}} = \frac{d/V}{L/R} = \frac{Rd}{VL}$$

where:

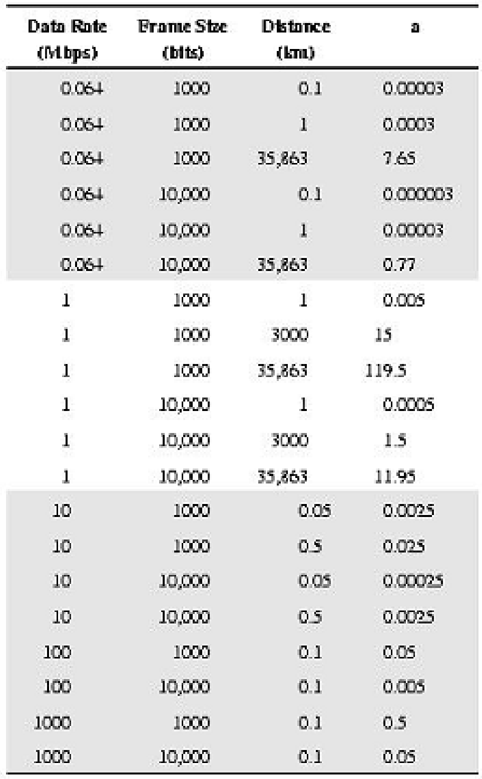
- d = distance between stations
- V = velocity of signal propagation
- L = length of frame in bits
- R = data rate on link in bits per sec



(a) $a > 1$ (b) $a < 1$

Figure 11.8 Stop-and-Wait Timing (transmission time = 1; propagation time = a)

Some Values of a



Data Rate (Mbps)	Frame Size (bits)	Distance (km)	a
0.064	1000	0.1	0.00003
0.064	1000	1	0.0003
0.064	1000	35,863	7.65
0.064	10,000	0.1	0.000003
0.064	10,000	1	0.00003
0.064	10,000	35,863	0.77
1	1000	1	0.005
1	1000	3000	15
1	1000	35,863	119.5
1	10,000	1	0.0005
1	10,000	3000	1.5
1	10,000	35,863	11.95
10	1000	0.05	0.0025
10	1000	0.5	0.025
10	10,000	0.05	0.00025
10	10,000	0.5	0.0025
100	1000	0.1	0.05
100	10,000	0.1	0.005
1000	1000	0.1	0.5
1000	10,000	0.1	0.05

Performance of Send and Wait

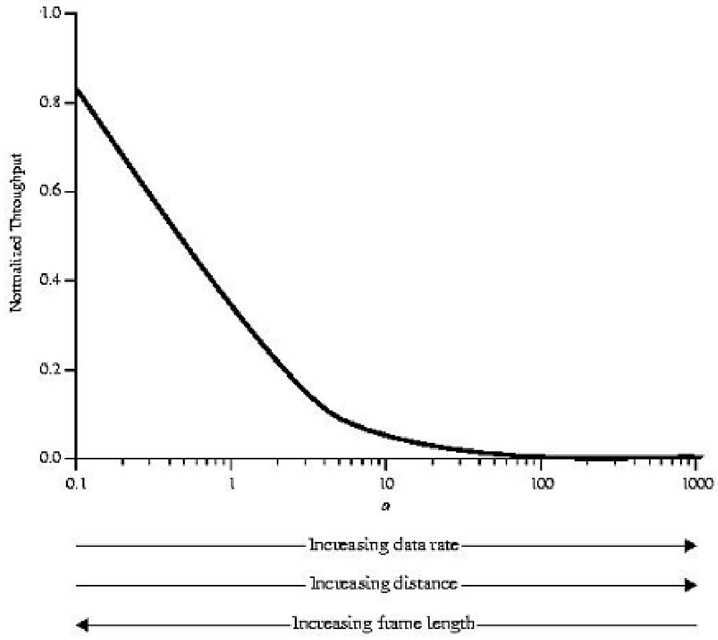
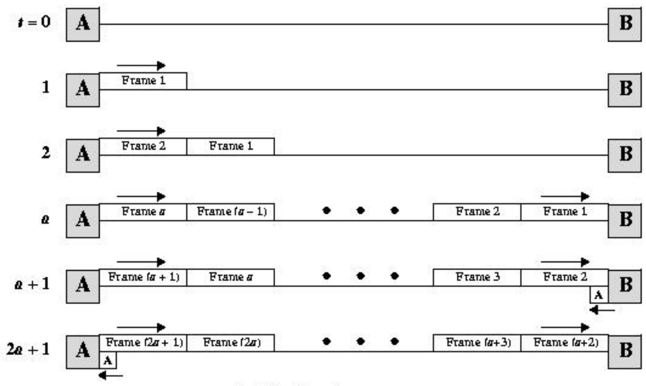


Figure 11.9 Performance of Stop-and Wait Protocol ($P = 10^{-3}$)

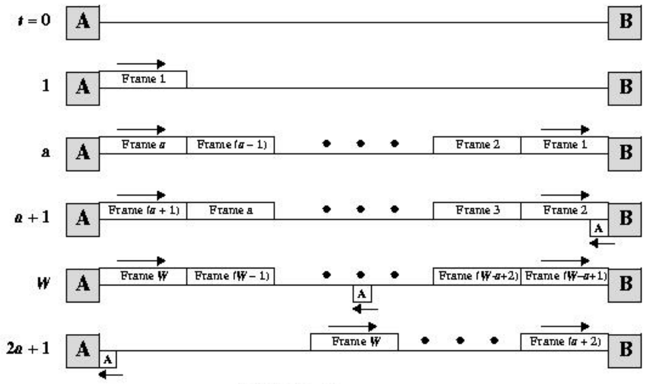
Performance: Error-Free Sliding Window ARQ (1)

- Let W be the number of frames that the sender can send, before he has to wait for an acknowledgement
  - It will be explained in a later lecture, why it is necessary to restrict the sender from sending arbitrary number of packets
- Case 1: $W \geq 2a + 1$
  - Ack for frame 1 reaches A before A has exhausted its window
- Case 2: $W < 2a + 1$
  - A exhausts its window at $t = W$ and cannot send additional frames until $t = 2a + 1$
- Normalized Throughput:

$$S = \begin{cases} 1 & W \geq 2a + 1 \\ \dfrac{W}{2a + 1} & W < 2a + 1 \end{cases}$$



(a) W 2a + 1



(b) W < 2a + 1

Figure 11.10 Timing of Sliding-Window Protocol

Performance: Sliding Window Utilization as Function of a

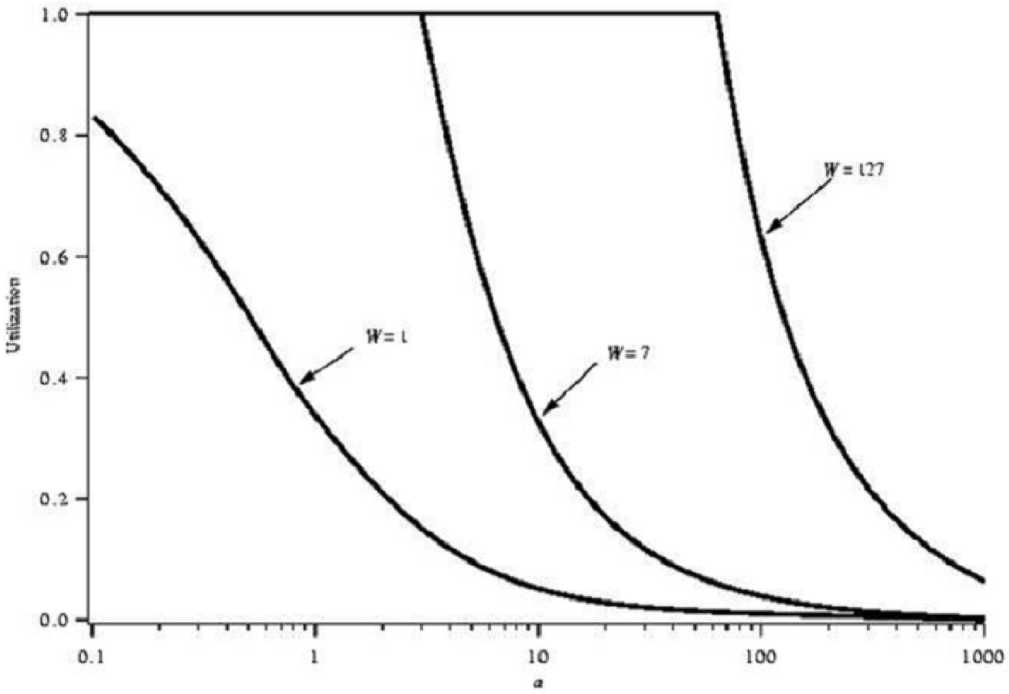


Figure 11.11 Sliding-Window Utilization as a function of *a*

Performance: Normalized Throughput in Case of Errors

- Selective Reject:

$$S = \begin{cases} 1 - P & W \geq 2a + 1 \\ \dfrac{W(1 - P)}{2a + 1} & W < 2a + 1 \end{cases}$$

- Go-Back-N:

$$S = \begin{cases} \dfrac{1 - P}{1 + 2aP} & W \geq 2a + 1 \\ \dfrac{W(1 - P)}{(2a + 1)(1 - P + WP)} & W < 2a + 1 \end{cases}$$

Performance: ARQ Utilization as Function of a

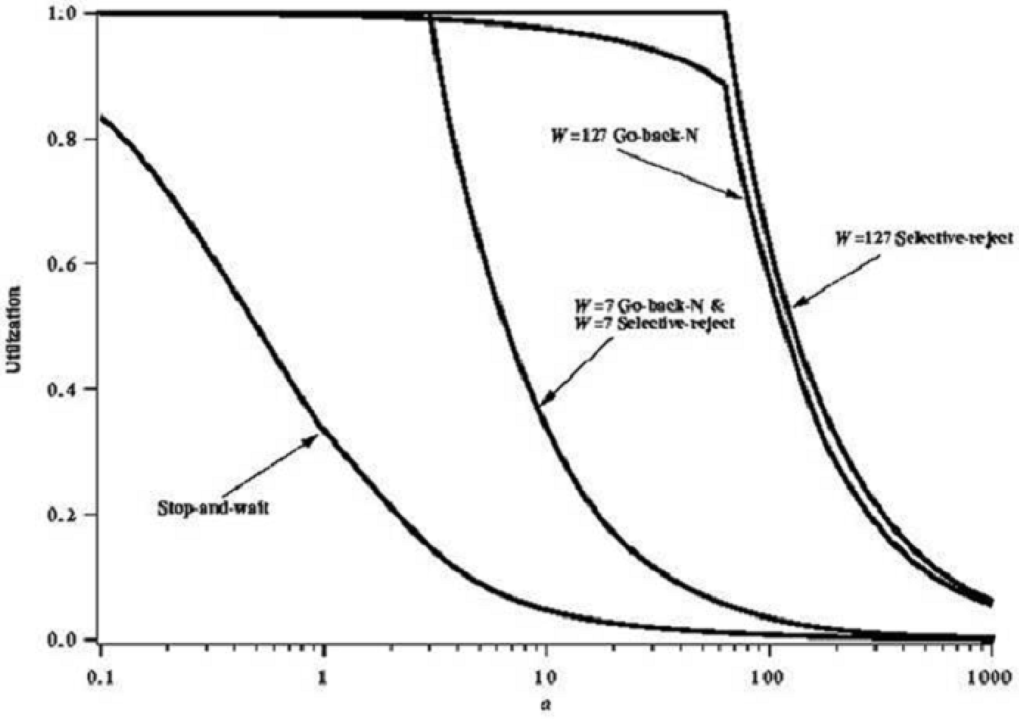


Figure 11.12 ARQ Utilization as a Function of *a* ($P = 10^{-3}$)

Performance: ARQ Utilization as Function of W

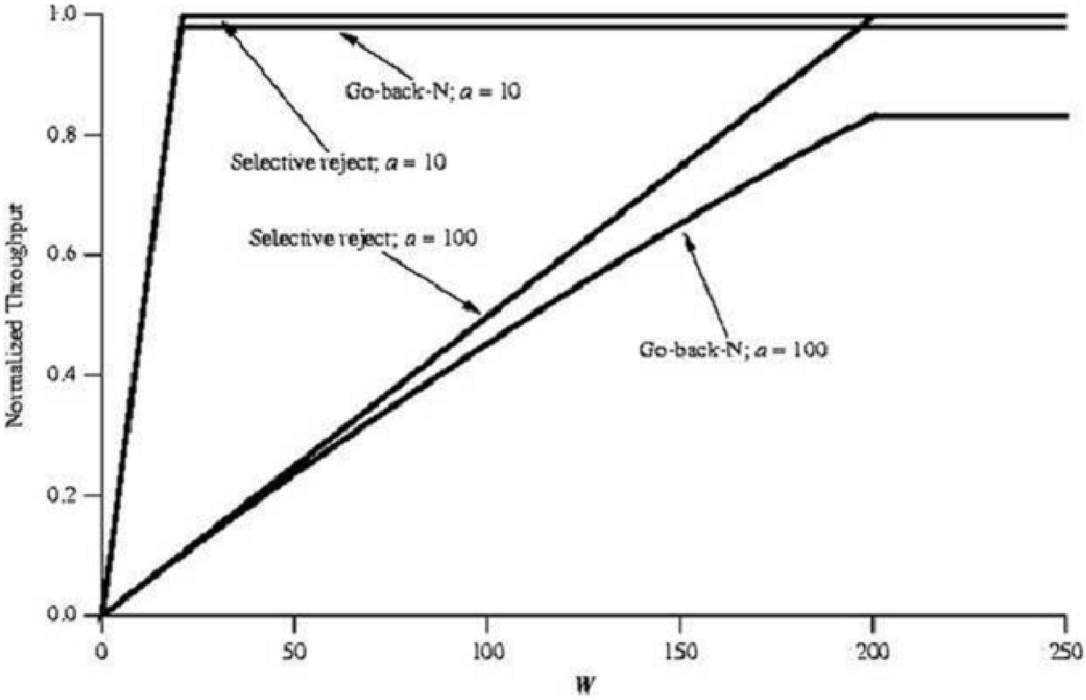


Figure 11.13 ARQ Throughput as a Function of *W* ($P = 10^{-3}$)

Conclusions

- 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
  - Or to detect them and repair them afterwards
- Flow control is often tightly integrated with error control (and sometimes also congestion control) in practical protocols
  - But it is a separate function and can be realized separately as well
- Choice of error control scheme (and its parameters) has implications on achievable performance
- Connection setup/teardown still has to be treated
  - Necessary to initialize a joint context for sender and receiver (e.g. initial sequence numbers, window size)

Additional References

[Sta04] W. Stallings. *Data and Computer Communications.* 7th edition, Prentice Hall, 2004.

[Tan02] A. S. Tanenbaum. *Computer Networks.* 4th edition, Prentice Hall, 2002.