Data Link Layer: Framing, Error Detection/Correction, and Protocol Mechanisms

1. Framing and Types

1.1 What is Framing?

Framing is a fundamental function of the Data Link Layer (DLL) that provides a way for a sender to transmit a set of bits that are meaningful to the receiver. The data link layer encapsulates packets received from the network layer into **frames** for transmission over the physical channel.

Key Components of a Frame:

- **Frame Header** - Contains physical (MAC) address of next receiving node
- **Payload Field** - Holds the packet from network layer
- **Frame Trailer** - Contains error detection/correction bits (checksum, CRC)

Why Framing is Necessary:

The physical layer transmits continuous bit streams. Without framing, it's impossible to distinguish where one message ends and another begins. Consider the postal system analogy: inserting a letter into an envelope separates one piece of information from another—the envelope serves as a delimiter.

1.2 Frame Detection Mechanism

Detecting Start of Frame:

- Every station listens to the link for a special **Starting Frame Delimiter (SFD)** pattern
- A sequential circuit detects this SFD and alerts the station
- Station checks destination address to accept or reject the frame

Detecting End of Frame:

- Stations must know when to stop reading the frame
- This is handled through various framing methods discussed below

1.3 Types of Framing

Type 1: Fixed-Size Framing

Characteristics:

- Frame is of fixed size with no boundary indicators needed
- Length of frame itself acts as a delimiter

Drawback: Internal fragmentation if data size < frame size

Solution: Padding (adding extra bits to make data fit the frame size)

Example: If frame size is 100 bits but data is 80 bits, add 20 padding bits

Type 2: Variable-Size Framing

Variable-size framing requires explicit boundary definitions. Three methods exist:

Method A: Length Field

Header	Length Field	Payload	Trailer
Address	**n**	Data (n bytes)	Checksum

Figure 1: Variable-Size Framing Using Length Field

How it works:

- A length field in frame header specifies the number of bytes in the frame
- Used in **Ethernet (802.3)**
- Receiver reads length field to know when frame ends

Problem: If length field itself gets corrupted, entire frame is lost

Method B: End Delimiter (ED)

How it works:

- An ED pattern is introduced to indicate end of frame
- Used in **Token Ring**

Problem: End delimiter can occur in the data itself, causing false frame termination

Method C: Flag Bytes with Byte Stuffing

A flag byte (8-bit delimiter, e.g., 01111110) marks frame boundaries.

Problem: Flag byte may occur in actual data, especially binary data (images, audio)

Solution: Byte Stuffing

- Sender's DLL inserts a special escape byte (ESC) before each accidental flag byte in data
- Receiver removes these escape bytes before giving data to network layer

Example:

- Flag byte: 01111110
- If data contains: 01111110, sender sends: ESC + 01111110
- Receiver sees ESC + 01111110, removes ESC, gets original: 01111110

Method D: Flag Bits with Bit Stuffing

Frames can contain arbitrary number of bits using bit-level framing.

How it works:

- Each frame begins and ends with special bit pattern: **01111110** (0x7E in hex)
- Whenever sender encounters **five consecutive 1s** in data, it automatically stuffs a **0 bit**
- Receiver sees five consecutive 1s followed by 0 bit and automatically de-stuffs (removes) the 0 bit

Example 1:

- Original data: 01111 (ends with five 1s)
- Data with bit stuffing: 011110 (0 bit inserted after five 1s)
- Receiver removes the stuffed bit to recover: 01111

Example 2:

- Original data: 011100011110
- ED pattern: 0111
- After bit stuffing: 0111**0**00011**0**110
- (0 bits stuffed after five consecutive 1s)

Advantages:

- Frames can have variable length
- No dependency on character boundaries
- Works with any bit stream including binary data

1.4 Problems and Solutions in Framing

Problem	**Drawback**	**Solution**
Fixed Size	Internal fragmentation	Padding
Length Field Corruption	Frame loss	Redundancy in length field
Flag Byte in Data	False termination	Byte stuffing (ESC insertion)
Flag Bits in Data	False termination	Bit stuffing (insert 0)

Table 1: Framing Problems and Solutions

2. Error Detection and Correction Algorithms

2.1 Introduction to Error Detection and Correction

The central concept in detecting or correcting errors is **redundancy**. To detect or correct errors, we send extra bits (redundant bits) with our data. These are added by sender and removed by receiver. Their presence allows the receiver to detect and/or correct corrupted bits.

Trade-off: More redundant bits = better error detection/correction capability, but lower channel efficiency

2.2 Parity Check

Simplest and least expensive error detection technique

Simple Parity Check

A k-bit dataword is changed to an n-bit codeword where **n** = **k** + **1**. The extra bit (parity bit) makes total number of 1s in codeword either even or odd.

Types:

Even Parity:

- Count number of 1s in dataword
- If count is odd, parity bit = 1 (to make total even)
- If count is even, parity bit = 0 (keep total even)

Odd Parity:

- If count is even, parity bit = 1 (to make total odd)
- If count is odd, parity bit = 0 (keep total odd)

Example (Even Parity):

- Dataword: 1011 (contains three 1s - odd)
- Parity bit: 1 (to make total four 1s - even)
- Codeword: 10111

Advantages:

- Detects all single-bit errors
- Simple to implement

Limitations:

- Cannot detect even-number bit errors
- Cannot correct errors (only detection)

Two-Dimensional Parity Check

Organizes data units into table of rows and columns. Parity bit calculated for each column and added as new row.

Advantages:

- Can detect burst errors better than simple parity
- Can correct single-bit errors in some cases

Drawback:

- More complex, requires more redundancy

2.3 Hamming Code (Error Correcting Code)

Hamming code is a set of error-correction codes developed by **R.W. Hamming** that can detect and correct single-bit errors.

Key Concepts

Redundant Bits (Parity Bits):

- Extra binary bits added to information-carrying bits
- Placed at calculated positions to eliminate errors
- **Hamming Distance:** Distance between two codewords = number of bit positions that differ

Calculating Number of Redundant Bits:

For M data bits and P redundant bits, total frame N = M + P

The following formula determines minimum P needed:

$$2^P \geq M + P + 1$$

Where:

- P = number of redundant bits
- M = number of data bits
- The "+1" accounts for no-error state

Example:

- For M = 4 data bits: need $2^P \geq 4 + P + 1 = 5 + P$
- If P = 2: $2^2 = 4$ (not enough)
- If P = 3: $2^3 = 8 \geq 5 + 3 = 8$ ✓ (P = 3 works)

Hamming Code Algorithm

Step 1: Position Assignment

- Write bit positions starting from 1 in binary form: 1, 10, 11, 100, 101, 110, 111, 1000...
- Mark positions that are powers of 2 (1, 2, 4, 8, 16, 32...) as parity bit positions
- All other positions (3, 5, 6, 7, 9, 10, 11, 12...) hold data bits

Step 2: Parity Bit Calculation

For even parity (set parity bit to make total 1s even):

Position 1 (P_1): Check 1 bit, skip 1 bit, check 1 bit, skip 1 bit...

- Checks positions: 1, 3, 5, 7, 9, 11, 13, 15...
- Formula: Check all positions where bit 0 (LSB) = 1 in binary representation

Position 2 (P_2): Check 2 bits, skip 2 bits, check 2 bits, skip 2 bits...

- Checks positions: 2, 3, 6, 7, 10, 11, 14, 15...
- Formula: Check all positions where bit 1 = 1

Position 4 (P_4): Check 4 bits, skip 4 bits, check 4 bits, skip 4 bits...

- Checks positions: 4, 5, 6, 7, 12, 13, 14, 15, 20, 21...
- Formula: Check all positions where bit 2 = 1

Position 8 (P_8): Check 8 bits, skip 8 bits...

- Checks positions: 8, 9, 10, 11, 12, 13, 14, 15, 24, 25...
- Formula: Check all positions where bit 3 = 1

Step 3: Set Parity Bit

For each parity bit position:

- Count number of 1s in positions it checks
- If count is odd, set parity bit = 1 (even parity)
- If count is even, set parity bit = 0 (even parity)

Hamming Code Example

Generate Hamming code for dataword: 1110

Solution:

Step 1: Position dataword bits

- M = 4 data bits, need P = 3 parity bits (from formula)
- Total N = 7 bits

Position layout:

- Position 1: P_1 (parity bit)
- Position 2: P_2 (parity bit)
- Position 3: $d_3 = 1$ (data bit 1 of 1110)
- Position 4: P_4 (parity bit)
- Position 5: $d_2 = 1$ (data bit 2)
- Position 6: $d_1 = 1$ (data bit 3)
- Position 7: $d_0 = 0$ (data bit 4)

Codeword so far: $P_1P_21P_4110$

Step 2: Calculate P_1 (checks positions 1,3,5,7)

- Positions: 1(P_1), 3(1), 5(1), 7(0)
- Count of 1s at these positions: 1 + 1 = 2 (even)
- Set $P_1 = 0$ (to keep even)

Step 3: Calculate P_2 (checks positions 2,3,6,7)

- Positions: 2(P_2), 3(1), 6(1), 7(0)
- Count of 1s: 1 + 1 = 2 (even)
- Set $P_2 = 0$

Step 4: Calculate P_4 (checks positions 4,5,6,7)

- Positions: 4(P_4), 5(1), 6(1), 7(0)
- Count of 1s: 1 + 1 = 2 (even)
- Set $P_4 = 0$

Final Hamming Code: 0010110

Verification: At receiver, calculate syndrome:

- $S_1 = P_1 \oplus d_3 \oplus d_2 \oplus d_0 = 0 \oplus 1 \oplus 1 \oplus 0 = 0$
- $S_2 = P_2 \oplus d_3 \oplus d_1 \oplus d_0 = 0 \oplus 1 \oplus 1 \oplus 0 = 0$
- $S_4 = P_4 \oplus d_2 \oplus d_1 \oplus d_0 = 0 \oplus 1 \oplus 1 \oplus 0 = 0$

Syndrome = 000 → No error detected ✓

2.4 Cyclic Redundancy Check (CRC)

CRC is based on polynomial division. In cyclic codes, if a codeword is cyclically shifted (rotated), the result is another valid codeword.

Example: If 1011000 is codeword, then 0110001 (cyclic left-shift) is also codeword.

CRC Process at Sender

Step 1: Frame Preparation

- Dataword has k bits (k = 4 in example)
- Augment dataword by adding (n-k) zeros to right
- n = total codeword bits, n-k = number of zeros to add

Step 2: Generator Division

- Use predefined CRC polynomial as divisor (size n-k+1)
- Example: Polynomial $x^3 + x + 1$ gives divisor 1011 (4 bits)
- Perform modulo-2 division on augmented dataword

Step 3: Reminder Appending

- Divide augmented dataword by divisor using modulo-2 division
- Append remainder (n-k bits) to original dataword
- Result is final codeword (n bits)

CRC Example: C(7,4)

7-bit codeword from 4-bit dataword

Dataword: 1011
CRC Polynomial: $x^3 + x + 1$ → Divisor: 1011

Sender Process:

1. Augment dataword: 1011000 (append three 0s)
2. Modulo-2 divide by 1011:
   1010

1011 | 1011000
1011

0000
0000

0000
0000

000

3. Remainder: 000
4. Codeword: 1011000

CRC Process at Receiver

Step 1: Syndrome Calculation

- Perform same modulo-2 division on received codeword
- Remainder is syndrome

Step 2: Error Detection

- If syndrome = 0 → No error, accept codeword
- If syndrome ≠ 0 → Error detected, discard codeword

Note: CRC can detect but not correct errors. Multiple bit errors within the code distance cannot be detected.

CRC Advantages and Applications

- Detects single and burst errors
- Good performance with long frames
- Used in modern protocols: Ethernet, Wi-Fi, cellular networks
- Hardware implementation efficient using shift registers

2.5 Checksum

Checksum is an error-detecting technique applicable to messages of any length, based on redundancy concept like CRC.

Checksum Algorithm

At Sender:

1. **Partition Message:** Divide message into fixed-size units (e.g., 4-bit numbers)
2. **Sum Calculation:** Add all units
3. **Wrap-Around:** If sum exceeds n bits, add overflow back to n rightmost bits (ones' complement addition)
4. **Complement:** Take ones' complement of sum to get checksum
5. **Transmission:** Send message units + checksum

Checksum Example

Message: Seven 4-bit numbers: 7, 11, 12, 0, 6

Step 1: Sum numbers

- 7 + 11 + 12 + 0 + 6 = 36

Step 2: Convert to 4-bit (ones' complement)

- 36 in binary: 100100
- Add overflow: 1 + 0100 = 0101 = 5

Step 3: Complement

- Ones' complement of 5: 1111 - 0101 = 1010 = 10
- **Checksum = 10**

Step 4: Send

- Numbers: 7, 11, 12, 0, 6
- Checksum: 10

At Receiver:

1. **Receive All Units:** 7, 11, 12, 0, 6, 10
2. **Sum with Ones' Complement:** 7 + 11 + 12 + 0 + 6 + 10 = 46
   - 46 in binary: 101110
   - Ones' complement addition: 1 + 01110 = 01111 = 15
3. **Check Result:**
   - If result = 1111...1 (all 1s) → No error
   - Otherwise → Error detected

Advantages:

- Simple to implement
- Fast computation
- Works with any message length

Limitations:

- Cannot correct errors
- Cannot detect all error patterns

Technique	Detection	Correction	Redundancy	Complexity
Simple Parity	Single bit	No	1 bit	Very Low
Hamming Code	Single bit	Single bit	$\log_2 M$	Low
CRC	Burst + Single	No	n-k bits	Medium
Checksum	Multiple	No	1 unit	Low

Table 2: Comparison of Error Detection and Correction Methods

---

3. Flow Control and ARQ Protocols

3.1 Introduction to Flow Control

Purpose: Ensure all frames are eventually delivered to network layer at destination in proper order.

Assumptions:

- Receiver can determine frame correctness using error detection/correction
- Physical, data link, and network layers engage in peer-to-peer communication
- Each layer has physically independent processor
- Machines can communicate in one or both directions simultaneously
- Sender has infinite data available

Flow Control Methods: Receiver sends feedback (acknowledgements) about incoming frames to sender

3.2 Stop-and-Wait Protocol

Simple point-to-point protocol where frames sent one at a time

How Stop-and-Wait Works:

1. **Sender Action:** Transmits one frame to destination
2. **Receiver Action:** Receives frame and checks for errors
3. **Receiver Response:**
   - If correct: Sends ACK (positive acknowledgement) with next expected frame number
   - If corrupted: Sends NAK (negative acknowledgement) or no response
4. **Sender Wait:** Waits for ACK before sending next frame
5. **Sender Decision:**
   - If ACK received: Send next frame
   - If NAK received or timeout: Retransmit same frame

Advantages:

- Simple to implement
- Works well for few large frames
- No buffering complexity at receiver

Disadvantages:

- **Very inefficient:** Sender idle while waiting for ACK
- **Poor link utilization** especially with long propagation delay
- Entire bandwidth wasted during waiting period

Link Utilization Problem:

For long frames and high propagation delays:

$$\text{Efficiency} = \frac{L/B_w}{L/B_w + 2 \times \text{Propagation Delay}}$$

Where:

- L = frame length (bits)
- B_w = bandwidth (bits/sec)
- Propagation delay = time for signal to travel from sender to receiver

Example: If transmission time = 1 second but propagation delay = 10 seconds:

- Efficiency = 1/(1 + 20) = 4.7%
- Sender idle 95.3% of time!

Stop-and-Wait Diagram:

Sender Receiver

Frame 1 ------>
<------ ACK (Frame 2)

Frame 2 ------>
<------ ACK (Frame 3)

(Sender waits here for ACK)

3.3 Sliding Window Flow Control

Allows multiple frames in transit simultaneously, dramatically improving efficiency

Basic Concept:

- Sender can transmit up to **W frames** without receiving ACK
- Receiver has buffer of size **W** (window size)
- Each frame is assigned **sequence number**
- ACK includes **next frame number expected** by receiver

Sequence Numbers:

- Frames numbered sequentially: 0, 1, 2, 3, ..., 2^k - 1, 0, 1, 2...
- Numbers are modulo 2^k (bounded by field size)
- If k-bit field used: sequence numbers cycle through 0 to 2^k - 1

Why Bounded?

- Receiver needs to know if received frame is new or retransmission
- Minimum field size: k = $\log_2$(W+1) bits

Sliding Window Operation:

At Sender:

- **Usent:** Frames already sent but not yet ACKed
- **Unsent:** Frames not yet sent
- Window = all frames that can be sent without ACK
- As ACK received, window "slides" forward

At Receiver:

- **Received in-order:** Frames delivered to network layer
- **Buffered out-of-order:** Frames received but waiting for preceding frames
- Window = range of frame numbers receiver can accept

Example: Window Size W = 4

Initial:
Sender sends frames 0, 1, 2, 3 (window size 4)

```
    |----Window (4 frames)----|
```

Frames: 0 1 2 3 4 5 6 7 0 1...
^
Sent

Receiver ACKs frame 2 (received 0,1,2):
|----Window slides---|
Frames: 0 1 2 3 4 5 6 7 0 1...
^
Next expected

Sender now sends frame 4 (window slides):
Frames: 0 1 2 3 4 5 6 7 0 1...
|----Window----|

Advantages:

- **High efficiency:** Multiple frames in transit
- **Better bandwidth utilization:** Sender transmits continuously
- **Flexible:** Can handle various propagation delays
- **Error recovery:** Retransmit specific bad frames

Disadvantage:

- **Complexity:** Need frame numbering, buffering, ACK tracking

3.4 Go-Back-N Protocol

ARQ Protocol using sliding window with retransmission strategy: discard all frames after error

How Go-Back-N Works:

When sender detects error/timeout:

1. **Retransmit** the bad frame
2. **Retransmit all subsequent frames** in sender's window
3. Receiver **discards all frames** after the bad one
4. Receiver waits for correct frame before accepting more

Go-Back-N Example:

Sender transmits frames: 0, 1, 2, 3, 4, 5, 6

```
Frame 0 ✓
Frame 1 ✓
Frame 2 ✗ (ERROR)
Frame 3 (received but discarded by receiver)
Frame 4 (received but discarded by receiver)
Frame 5 (received but discarded by receiver)
```

Receiver sends: "Need frame 2"

Sender goes back N positions and retransmits:
Frame 2 (retransmitted)
Frame 3 (retransmitted)
Frame 4 (retransmitted)
Frame 5 (retransmitted)
...

Key Parameters:

- **Window size W:** Determines how many frames sender can send before ACK
- **Sequence number field:** k bits allow numbering 0 to 2^k - 1
- **Constraint:** W ≤ 2^(k-1) (prevent confusion with retransmissions)

Advantages:

- Simple implementation
- Uses standard sliding window
- Good for low error rate channels

Disadvantages:

- **Inefficient on high error rate channels:** Retransmits many correct frames
- **Wastes bandwidth:** Throws away correctly received frames
- Many unnecessary retransmissions in error-prone links

Go-Back-N State Machine:

Sender:
State: SEND

- Transmit frame (if window not full)
- Increment send counter
- Start timer

On ACK: Update send window
On TIMEOUT: Retransmit from bad frame onward

Receiver:
State: RECEIVE

- Accept only in-sequence frames
- Send ACK for last in-order frame received
- Discard all out-of-sequence frames

3.5 Selective Repeat (Selective Stop-and-Wait) Protocol

ARQ Protocol that retransmits ONLY the bad frame, not subsequent frames

Key Difference from Go-Back-N:

Aspect	Go-Back-N	Selective Repeat
Error Response	Retransmit bad + all after	Retransmit ONLY bad frame
Receiver Buffer	Window = 1	Window > 1 (stores out-of-order)
Memory Requirement	Minimal	Large (buffers many frames)
Efficiency	Lower on error-prone channels	Higher on error-prone channels
Implementation	Simple	Complex

How Selective Repeat Works:

1. **Sender transmits** multiple frames (up to window size W)
2. **Receiver has buffer W** to store frames
3. **On error:** Receiver buffers frame in correct position
4. **Sender retransmits ONLY** the bad frame
5. **Receiver may accept frames out-of-order** and buffer them
6. **When preceding frames arrive:** Deliver sequence to network layer

Selective Repeat Example:

Initial transmission:
Sender sends: 0, 1, 2, 3, 4, 5

Receiver status:
✓ Frame 0 (delivered)
✓ Frame 1 (delivered)
✗ Frame 2 (ERROR)
? Frame 3 (buffered, waiting for 2)
? Frame 4 (buffered, waiting for 2)
? Frame 5 (buffered, waiting for 2)

Sender receives: "Resend frame 2"
Sender retransmits: Frame 2 ONLY

Receiver gets frame 2:
✓ Frame 2 (now have 2,3,4,5 in sequence)
All buffered frames delivered to network layer

Constraints for Selective Repeat:

Window Size Constraint: $W \leq 2^{k-1}$

Where k = number of bits for sequence number

Why? Without this constraint:

- Old retransmissions could be confused with new transmissions
- Example (k=2, so 0,1,2,3 sequence numbers):
    - If W=4: After sending 0,1,2,3 and retransmitting, new 0 could match old 0
    - Receiver can't distinguish new frame 0 from retransmitted old frame 0

Advantages:

- **Efficient:** Retransmits only bad frames
- **Good for error-prone channels:** Reduces wasted bandwidth
- **Faster recovery:** Don't retransmit correct frames

Disadvantages:

- **Complex implementation:** Need frame buffering logic
- **Memory overhead:** Receiver must buffer multiple frames
- **Complex sender/receiver logic:** Track multiple frames

Comparison Table:

Protocol	Complexity	Efficiency	Best Use
Stop-and-Wait	Very Low	Very Low	Low-speed, short distances
Go-Back-N	Low	Medium	Low error rate channels
Selective Repeat	High	High	High error rate channels

Table 3: ARQ Protocol Comparison

3.6 Performance Analysis

Stop-and-Wait Efficiency:

$$Efficiency = \frac{T_f}{T_f + 2T_p}$$

Where:

- T_f = frame transmission time
- T_p = propagation delay

Example Calculation:

- Frame size: 10,000 bits
- Bandwidth: 1 Mbps
- Propagation delay: 45 ms

$T_f = 10,000/1,000,000 = 0.01$ s = 10 ms
$Efficiency = 10/(10 + 2 \times 45) = 10/100 = 10\%$

Only 10% efficiency! 90% time wasted waiting.

Sliding Window Efficiency:

$$Efficiency = \min(1, \frac{W \times T_f}{T_f + 2T_p})$$

With W = 10 in same example:
$Efficiency = 10 \times 10/100 = 100/100 = 1 = 100\%$

Full efficiency possible!

Summary and Key Takeaways

Framing:

- Essential for separating messages in bit stream
- Fixed-size uses padding, variable-size uses length fields, delimiters, or bit stuffing
- Bit stuffing most versatile for arbitrary binary data

Error Detection/Correction:

- **Parity:** Simple detection, cannot correct
- **Hamming Code:** Detects and corrects single-bit errors using formula $2^P \geq M + P + 1$
- **CRC:** Polynomial-based, detects burst errors, no correction
- **Checksum:** Simple, works for any length, no error correction

Flow Control Protocols:

- **Stop-and-Wait:** Simple but inefficient (10-20% utilization)
- **Go-Back-N:** Better efficiency, simple, but wastes bandwidth on errors
- **Selective Repeat:** Best efficiency for error-prone channels, complex implementation

Exam Tips:

1. Know frame components (header, payload, trailer) and why each is needed
2. Understand bit stuffing mechanism with concrete examples
3. Hamming code formula and position calculation are frequently asked
4. CRC polynomial division process with modulo-2
5. Why selective repeat better than go-back-N on noisy channels
6. Calculate protocol efficiency given transmission and propagation times