Chapter-3: The Data Link Layer
[chapter-3: 3.1 to 3.4]

3.1 DATALINK LAYER DESIGN ISSUES

- The **data link layer** uses services of the **physical layer** to send and receive bits over communication channels, which may be unreliable and prone to data loss.
- **Key functions of the data link layer include:**
  - Providing a **well-defined service interface** to the network layer.
  - **Framing** sequences of bytes into self-contained segments (frames).
  - **Detecting and correcting errors** in transmission.
  - **Regulating data flow** to ensure slow receivers are not overwhelmed by fast senders.
- To achieve these functions, the data link layer:
  - Takes packets from the network layer.
  - **Encapsulates** them into frames for transmission.
  - Each frame consists of:
    - **Frame header**
    - **Payload** (containing the network-layer packet)
    - **Frame trailer**
  - Frame management is central to the operations of the data link layer.
  - On unreliable wireless networks, strong data link protocols significantly **improve overall performance**.

3.1 DATALINK LAYER DESIGN ISSUES

3.1.1 Services Provided to the Network Layer

- The data link layer's primary role is to **provide services to the network layer**.
- Its core service is to **transfer network-layer data** from the source machine to the destination machine.
- On the source side:
  - A **network-layer process** passes packets down to the data link layer for transmission.
- The data link layer:
  - **Transmits these packets** across the link to the destination machine.
  - Delivers them upward to the network layer there.
- Logically:
  - We view the communication as occurring between **two data link layer processes** communicating via a **data link protocol** (Fig. 3-2(a)).
- Physically:
  - The actual transmission occurs over the physical medium (Fig. 3-2(b)), but the conceptual model simplifies understanding.

3.1 Data Link Layer Design Issues

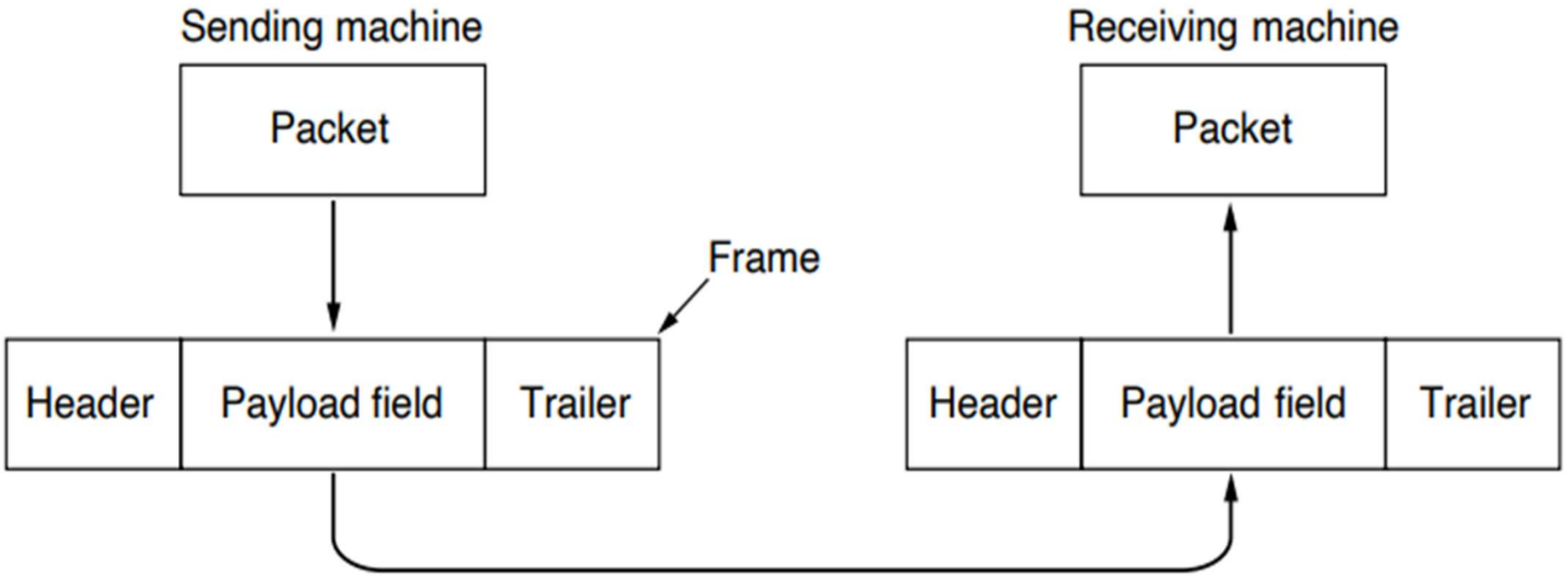

Figure 3-1. Relationship between packets and frames.

3.1.1 Services Provided to the Network Layer

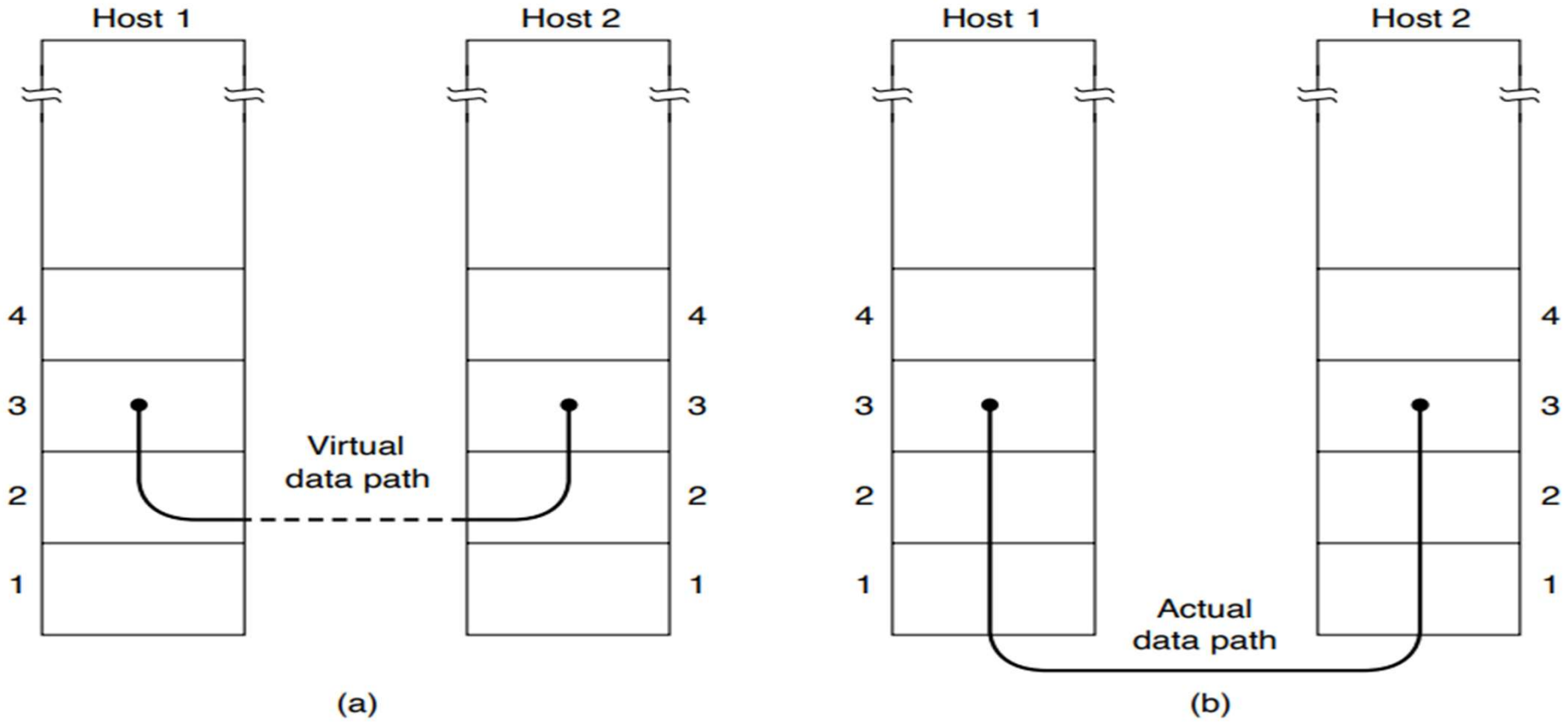

Figure 3-2. (a) Virtual communication. (b) Actual communication.

3.1 DATALINK LAYER DESIGN ISSUES

3.1.1 Services Provided to the Network Layer — Bullet Points

- The data link layer can be designed to offer several types of services depending on the protocol.
- Three commonly used service types are:
    1. **Unacknowledged connectionless service**
    2. **Acknowledged connectionless service**
    3. **Acknowledged connection-oriented service**

1. Unacknowledged Connectionless Service

- The source machine sends **independent frames** to the destination **without any acknowledgements**.
- **No logical connection** is established before or after transmission.
- **Lost frames are not detected or retransmitted** by the data link layer.
- Suitable when:
    - The **error rate is very low**, so higher layers handle recovery if needed.
    - For **real-time applications** (voice, video), where *late data is worse than incorrect data*.
- **Example:** Ethernet.

2. Acknowledged Connectionless Service

- Still **no logical connection**, but **each frame** is individually **acknowledged**.
- Sender knows whether:
    - A frame arrived safely, or
    - It was lost and must be retransmitted.
- Helps ensure reliable delivery over **unreliable channels** (e.g., noisy or wireless links).
- **Example:** IEEE 802.11 (Wi-Fi).
- Note:
    - Acknowledgements at the data link layer are **an optimization**, not a necessity.
    - The **network layer** can always implement its own acknowledgement and retransmission mechanism if needed.

3.1 DATALINK LAYER DESIGN ISSUES

3.1.1 Services Provided to the Network Layer — Bullet Points

Why Simple Network-Layer Acknowledgements Can Be Inefficient

- Network links often have:
  - A **maximum frame size** enforced by hardware.
  - **Known propagation delays**.

- The **network layer is unaware** of these parameters.
- A large packet from the network layer may be broken into many frames (e.g., 10).
  - If **2 frames are lost**, the whole packet must be retransmitted.
  - This leads to **significant delay**.

- If **individual frames** are acknowledged and retransmitted:
  - Errors are corrected **locally and faster**.

- On **highly reliable channels** (e.g., optical fiber), a complex data link protocol adds unnecessary overhead.
- On **unreliable channels** (e.g., wireless), this overhead is **worth it** for improved reliability.

3. Acknowledged Connection-Oriented Service

- The **most sophisticated** data link layer service.
- A **connection is established** between source and destination *before* data transfer begins.
- Each frame is:
  - **Numbered**, and
  - **Guaranteed to be delivered** by the data link layer.
- Guarantees provided:
  - Each frame is received **exactly once**.
  - All frames arrive **in order**.
  - No duplicates, no reordering.
- Provides the network layer with the equivalent of a **reliable bit stream**.
- Appropriate for:
  - **Long-distance**, unreliable links.
  - Example: **Satellite links**, long telephone circuits.
- If only acknowledged connectionless service were used:
  - Lost acknowledgements could cause **duplicate transmissions**, wasting bandwidth.

Phases of Connection-Oriented Service

1. **Connection Establishment**
   - Both machines initialize necessary:
     - Variables
     - Counters
     - State information for frame tracking
2. **Data Transfer**
   - One or more **numbered frames** are transmitted over the established connection.
3. **Connection Release**
   - Variables, buffers, and other allocated resources are **freed**.
   - The logical connection is terminated cleanly.

3.1 DATALINK LAYER DESIGN ISSUES

Need for Framing

- To provide services to the network layer, the data link layer must rely on the physical layer.
- The **physical layer**:
  - Sends a **raw bit stream**.
  - Adds some **redundancy** to reduce errors (especially on noisy channels).
- Despite this, the bit stream received at the data link layer:
  - **May contain errors** (bit flips).
  - May have **more, fewer, or the same number** of bits as transmitted.
- It is the **data link layer's responsibility** to detect—and if needed, correct—errors.

How Framing Works

- The data link layer breaks the continuous bit stream into **discrete frames**.
- For each frame:
  - A **checksum** (error-detection token) is computed and attached.
- At the receiver:
  - A **new checksum** is computed from the received data
  - If it **differs** from the transmitted checksum:
    - The frame contains an error.
    - The data link layer may **discard the frame** and/or **send an error report**.

Challenges in Framing

- Designing framing is harder than it seems.
- A good design must:
  - Allow the receiver to **easily recognize frame boundaries**.
  - Use **minimal channel bandwidth** for marking frames.
- Four major framing methods are commonly used.

Four Framing Methods

1. **Byte Count**
   - A field in the frame header specifies the **number of bytes** in the frame.
   - The receiver:
     - Reads the byte count.
     - Knows **exactly where the frame ends**.
   - Example frame sizes (as shown in Fig. 3-3(a)):
     - 5 bytes, 5 bytes, 8 bytes, and 8 bytes.
2. **Flag bytes with byte stuffing**
3. **Flag bits with bit stuffing**
4. **Physical layer coding violations**

3.1.2 Framing

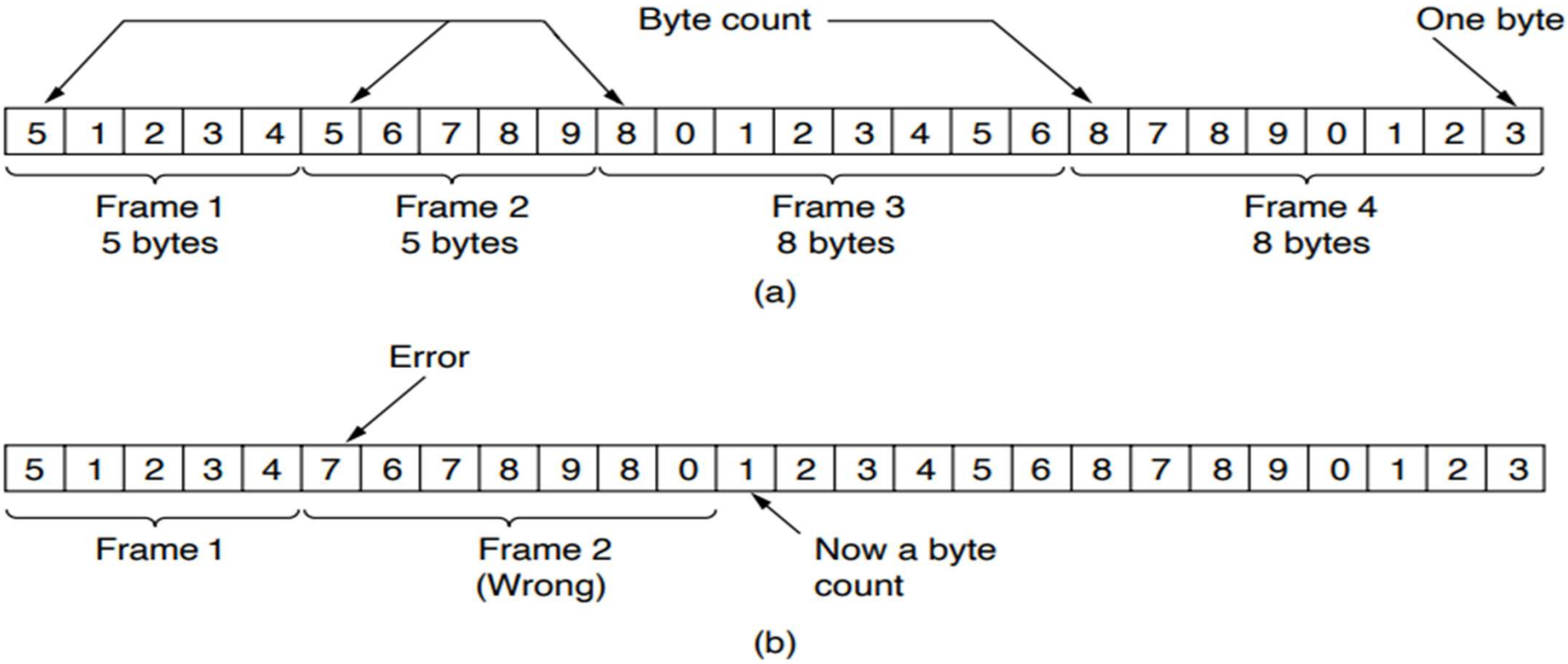

Figure 3-3. A byte stream. (a) Without errors. (b) With one error.

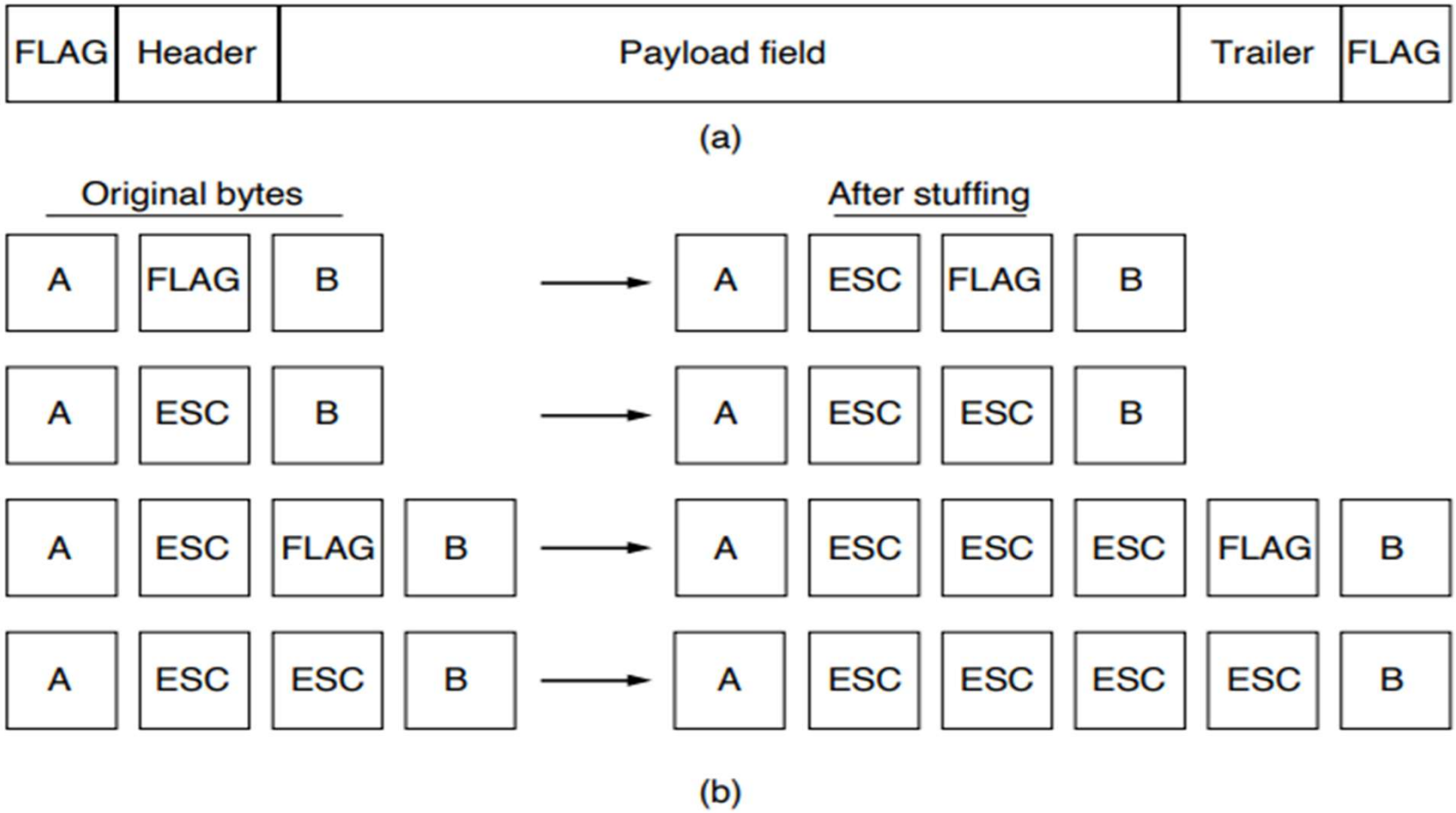

Figure 3-4. (a) A frame delimited by flag bytes. (b) Four examples of byte sequences before and after byte stuffing.

3.1 DATALINK LAYER DESIGN ISSUES

3.1.2 Framing

Problems with the Byte Count Method

- The **byte count field** can be corrupted by transmission errors.
  - Example: A frame with byte count **5** may be received as **7** due to a bit flip.
- If the count is wrong:
  - The receiver **loses frame boundary synchronization**.
  - The receiver cannot identify where the next frame begins.
- Even if the **checksum indicates an error**, the receiver still cannot determine:
  - How many bytes to skip,
  - Where the next frame actually starts.
- Requesting retransmission does not help because the receiver still cannot locate the proper boundary.
- Therefore, **byte count is rarely used alone**.

Flag Bytes and Byte Stuffing

Flag Byte Method

- Frames begin and end with a **special flag byte** (e.g., **FLAG**).
- Often the **same byte** marks both the start and end.
- **Two consecutive flag bytes** indicate:
  - End of one frame
  - Start of the next

If synchronization is lost, the receiver can simply **scan for two flag bytes** to resynchronize.

Problem

- The **flag byte may appear in the data**, especially with binary files (images, audio).
- This causes confusion because the receiver might mistake data for a frame boundary.

Byte Stuffing (with ESC Byte)

- To prevent false frame boundaries, the sender:
  - Inserts an **escape byte (ESC)** before every occurrence of the **flag byte** in the data.
- The receiver:
  - Removes ESC bytes before delivering data to the network layer.

Handling ESC in Data

- If the **ESC byte itself** appears in the data:
  - It is **also escaped** using another ESC.
- At the receiver:
  - The first ESC is removed.
  - The following byte (ESC or FLAG) is treated as data.

Key Properties

- After **destuffing**, the delivered data sequence is **exactly the same** as the original.
- Searching for frame boundaries:
  - Receiver only needs to detect **two consecutive FLAG bytes**.
  - No need to remove escapes during boundary detection.

Usage

- This byte-stuffing mechanism is a simplified version of what is used in:
  - **PPP (Point-to-Point Protocol)**
  - A widely-used protocol for carrying packets over communication links on the Internet.

3.1 DATALINK LAYER DESIGN ISSUES

3.1.2 Framing

Bit Stuffing (Third Framing Method)

- Bit stuffing solves a limitation of **byte stuffing**, which depends on 8-bit bytes.
- Bit-level framing allows:
  - Frames of **arbitrary length**.
  - Data units of **any size** (not necessarily bytes).
- Used in **HDLC (High-level Data Link Control)**.
- **Flag pattern** marking start and end of frames:
  - `01111110` (hex **0x7E**)
  - Called the **flag byte** (even though it's a bit pattern).

How Bit Stuffing Works

- Whenever the sender detects **five consecutive 1s** in data:
  - It automatically **inserts a 0** (stuffed bit).
- Purpose:
  - Prevents accidental appearance of the flag pattern in data.
  - Ensures adequate transitions for physical-layer synchronization.
- Receivers perform **destuffing**:
  - When they see **five 1s followed by a 0**, they **remove** the extra 0.
- Bit stuffing is **transparent** to upper layers:
  - If user data contain the flag pattern (01111110):
    - It is transmitted with stuffing (e.g., 011111010).
    - It is stored at the receiver as the original pattern (01111110).
- **USB** also uses bit stuffing for clock recovery.

Advantages

- Receiver can **unambiguously detect** frame boundaries:
  - Flag pattern **only appears at the start and end** of frames.
- If synchronization is lost, scanning for the flag suffices.

3.1 DATALINK LAYER DESIGN ISSUES

Frame Length Variability (for Byte & Bit Stuffing)

- Frame size depends on data content.
- Examples:
  - If no flag bytes in data → 100 bytes stays ~100 bytes.
  - If data = all flag bytes → each flag must be escaped → frame ~200 bytes.
  - With bit stuffing → approx **12.5% expansion** (1 stuffed bit every 8 bits).

Coding Violations (Fourth Framing Method)

- Some physical layer encodings contain **redundancy**, meaning:
  - Not all bit patterns are legal.
  - Illegal patterns can be used as **frame delimiters**.

Example: 4B/5B Line Coding

- Maps **4 data bits → 5 signal bits**.
- Out of 32 possible 5-bit values:
  - Only **16 are valid** (for actual data).
  - Remaining **16 unused values** can serve as **special markers**.
- Use of these invalid patterns to mark frame boundaries is called:
  - **Coding violations**.
- No need for bit/byte stuffing because:
  - Reserved patterns never appear in real data.

Combined Approaches (Used in Real Protocols)

- Many protocols combine multiple techniques for reliability.

- Example: **Ethernet and 802.11**
  - Frames begin with a **preamble**:
    - A long, well-defined sequence (e.g., **72 bits** in 802.11).
    - Helps the receiver synchronize before receiving actual data.
  - After preamble:
    - A **length field** indicates how many bytes follow, marking frame end.

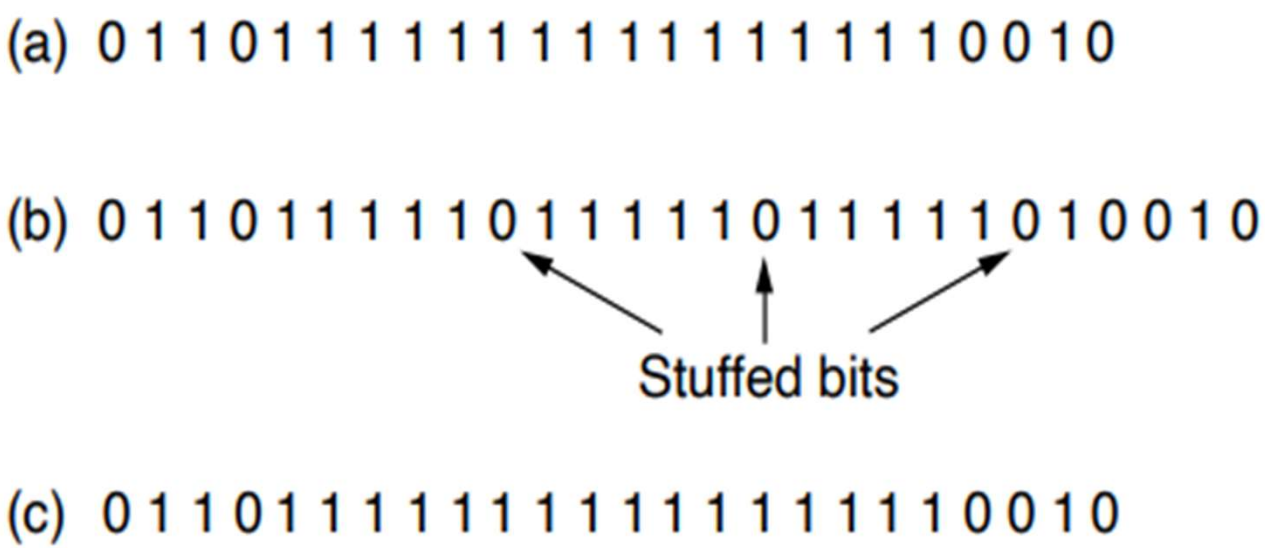

Figure 3-5. Bit stuffing. (a) The original data. (b) The data as they appear on the line. (c) The data as they are stored in the receiver's memory after destuffing.

3.1 DATA LINK LAYER DESIGN ISSUES

3.1.3 Error Control

Purpose of Error Control

- After framing is solved, the next task is ensuring:
  - All frames are **delivered reliably**,
  - Delivered **in correct order**,
  - Delivered **exactly once** to the network layer.
- Receiver is assumed capable of detecting whether a frame is:
  - **Correct**, or
  - **Corrupted** (via checksum or other mechanism).

Need for Feedback (Acknowledgements)

- Reliable, connection-oriented service **cannot** allow frames to be sent unchecked.
- To ensure reliability, the receiver sends **feedback** to the sender in the form of:
  - **Positive acknowledgements (ACKs)** → frame received safely.
  - **Negative acknowledgements (NAKs)** → frame damaged or faulty; retransmission required.

Problem: Lost Frames and Lost Acknowledgements

- Frames may **vanish** entirely (noise burst, hardware error).
  - Receiver will not acknowledge because it never saw the frame.
- Acknowledgement frames may also be **lost**.
  - Sender then does not know whether:
    - The frame was received or
    - The ACK was lost.
- A protocol where sender **waits indefinitely** for an ACK/NAK will **hang forever** if anything is lost.

3.1 DATA LINK LAYER DESIGN ISSUES

Solution: Timers

- Sender starts a **timer** whenever it sends a frame.
- Timer duration is long enough for:
  - Frame to reach the receiver,
  - Receiver to process it,
  - ACK to return to the sender.

- If ACK arrives:
  - Timer is **cancelled**.

- If no ACK is received before timer expires:
  - Timer **expires** → sender assumes a problem.
  - Frame is **retransmitted**.

Problem: Duplicate Frames

- Retransmissions may lead the receiver to receive the same frame **multiple times**.
- Without protection, the receiver might pass duplicates **multiple times** to the network layer.

Solution: Sequence Numbers

- Every outgoing frame is given a **sequence number**.

- Receiver uses sequence numbers to:
  - Distinguish **originals** from **retransmissions**.
  - Ensure each frame is delivered **exactly once**.

Key Role of Data Link Layer

- Manages:
  - **Timers**
  - **Sequence numbers**
  - **Retransmission logic**
- Ensures reliable, in-order, duplicate-free delivery to the network layer.

3.1 DATA LINK LAYER DESIGN ISSUES

3.1.4 Flow Control

- **Flow control** is needed when a **sender transmits frames faster than the receiver can process them**.
    - Common example: a powerful web server sending data to a slow smartphone.
- Even if no transmission errors occur, a slow receiver may **drop frames** if they arrive too quickly.
- **Two main approaches** to handle this issue:

  1. Feedback-based Flow Control
    - Receiver sends feedback to the sender.
    - Feedback tells sender whether it can continue sending or must slow down.
    - Used in both **link layer** and **higher layers**.
    - Many modern systems rely on higher-layer flow control while link-layer hardware runs at **wire speed** (i.e., fast enough not to be the bottleneck).

- **2. Rate-based Flow Control**
    - No feedback is used.
    - Protocol itself limits the sender's transmission rate.
- In feedback-based methods, the protocol defines **rules about when a sender may transmit the next frame**.
    - Often the sender must **wait for permission** (explicit or implicit).
    - Example: Receiver allows the sender to send **n frames**, and after that the sender must wait for further permission.

3.2 ERROR DETECTION AND CORRECTION

Error Handling Strategies — Bullet Points

- There are **two fundamental strategies** for managing transmission errors, both involving **redundant bits** added to transmitted data.
- **Strategy 1: Error-Correcting Codes (FEC – Forward Error Correction)**
    1. Enough redundancy is added so the receiver can **deduce the original data** even if errors occur.
    2. Preferred on **noisy channels** (e.g., wireless links) where **retransmissions are likely to fail** as well.
    3. Useful when the error rate is high and requesting retransmission is inefficient.
- **Strategy 2: Error-Detecting Codes**
    1. Adds just enough redundancy for the receiver to **detect an error**, not correct it.
    2. Receiver requests a **retransmission** when an error is found.
    3. Suitable for **highly reliable channels** (e.g., optical fiber), where errors are rare and retransmission is cheap.
- **Neither strategy can correct every possible error**, because redundant bits can also be corrupted during transmission.
- The channel does **not treat data bits and redundant bits differently**—both can get corrupted.
- Therefore, codes must be chosen to **handle the expected error patterns** of the channel.
- **Two common error models:**
    1. **Isolated Single-Bit Errors**
        - Often caused by occasional spikes of thermal noise.
    2. **Burst Errors**
        - Errors occur in clusters rather than singly.
        - Caused by deep wireless fades or electrical interference.
        - Have both benefits and drawbacks compared to single-bit errors.

3.2 ERROR DETECTION AND CORRECTION

Strategies for Handling Errors (Error Detection vs Error Correction)

- Two main strategies for dealing with transmission errors:
  - **Error-correcting codes (FEC)**: add enough redundancy so the receiver can **reconstruct the original data**.
  - **Error-detecting codes**: add redundancy only to let the receiver **detect an error** and request a retransmission.
- **FEC (Forward Error Correction)** is preferred for **noisy channels** (e.g., wireless) because retransmissions are likely to fail as well.
- **Error-detecting codes** are preferred for **highly reliable channels** (e.g., fiber optics), where retransmissions are rare and inexpensive.
- Redundant bits suffer errors just like data bits—**channels do not treat them differently**.
- Therefore, codes must be **strong enough** to handle the expected error patterns.
- Two major models of errors:
  - **Single-bit random errors** caused by occasional noise spikes.
  - **Burst errors** caused by fades, interference, or electrical disturbances.
- Both error models are important and involve **different trade-offs**.

3.2 ERROR DETECTION AND CORRECTION

3.2.1 Error-Correcting Codes

Overview

- Error-correcting codes add **redundant bits** to the transmitted data so that the receiver can **detect and correct** errors without retransmission.
- A transmitted frame consists of:
  - **m data bits** (message)
  - **r redundant bits** (check bits)
- Total bits: **n = m + r**, forming an **(n, m) code**.
- A transmitted unit of n bits is called a **codeword**.
- **Code rate = m / n** → Higher rate means less redundancy.

Types of Error-Correcting Codes Discussed

1. **Hamming codes**
2. **Binary convolutional codes**
3. **Reed–Solomon codes**
4. **Low-Density Parity Check (LDPC) codes**

Properties of Codes

1. Block Codes

- r check bits are generated **only from the m data bits**.
- Think of it as mapping data bits to check bits using a lookup table.

2. Systematic Codes

- Original **m data bits appear unmodified** in the codeword. Redundant bits appended separately.

3. Linear Codes

- r check bits computed as a **linear function** of data bits. XOR (mod-2 addition) commonly used.
- Enables implementation with:
  - Matrix multiplications
  - Simple logic circuits

3.2 ERROR DETECTION AND CORRECTION

Code Rate

- Fraction of useful information in the codeword: **m / n**.
- Examples:
  - **Rate = 1/2** → very noisy channels (half bits are redundancy)
  - **Rate ≈ 1** → reliable channels (few check bits)

Understanding Errors

- Errors are measured by the **number of bit differences** between two codewords.

- Example:
  Codeword1 = `10001001`
  Codeword2 = `10110001`

- To count differences:
  - XOR them
  - Count number of **1s** in the result → number of differing bits

Key Idea

- The ability of a code to detect or correct errors depends on:
  - How far apart valid codewords are in terms of **bit differences** (Hamming distance).
- Larger distances → stronger error protection.

3.2 ERROR DETECTION AND CORRECTION

Error-Correcting Codes & Hamming Distance

The **Hamming distance** between two codewords is the number of bit positions in which they differ.
Example:

```
10001001
10110001  → differ in 3 positions
```

- The Hamming distance matters because:
  - If two valid codewords differ by *d* bits, it takes *d* single-bit errors to convert one into the other.
  - The **minimum Hamming distance** of a code determines its **error-detecting and error-correcting power**.
- In block codes:
  - Not all possible 2^m bit patterns are valid codewords.
  - With mmm data bits and rrr check bits, only a fraction 2^m/2^n or ½^r of all possible bit combinations are valid.
  - The sparsity of valid codewords allows error detection/correction: illegal patterns reveal errors.
- Error capabilities:
  - To **detect** up to **d errors**, the code must have **minimum distance = d + 1**.
  - To **correct** up to **d errors**, the code must have **minimum distance = 2d + 1**.
    - This ensures that even if errors occur, the corrupted word is closer to the original than to any other valid codeword.
- Example:

A code with the four codewords:

```
0000000000
0000011111
1111100000
1111111111
```

  - These are widely spaced in Hamming distance, allowing correction of multiple-bit errors.

3.2 ERROR DETECTION AND CORRECTION

Error-Correcting Codes

- The illustrated code example has a **Hamming distance of 5**.
  - This means:
    - It can **correct up to 2-bit errors**.
    - It can **detect up to 4-bit errors**.
- **Example:**
  - Received codeword: `0000000111`
  - If the system assumes only **single- or double-bit errors**, the closest valid codeword is:
    - `0000011111`
  - Therefore, the receiver concludes that this was the original codeword.
- **But**, if a **triple-bit error** converts:
  - `0000000000` → `0000000111`
  - The receiver will incorrectly correct this to `0000011111`, leading to a wrong result.
    - Triple errors fall outside the correction capability.
- If we assume **all types of errors** (double, triple, quadruple), then:
  - The received codeword is **not a valid codeword**, so:
    - We can **detect an error**, but **cannot correct it** uniquely.
  - A code **cannot simultaneously**:
    - Correct double errors **and**
    - Detect quadruple errors,
    - Because that would require interpreting the same received word in multiple ways.

3.2 ERROR DETECTION AND CORRECTION

- **Decoding strategy:**
  - In simple examples, decoding can be done by visual inspection or comparing distances.
  - In general, decoding requires checking the distance to **all** codewords.
    - This can be **slow** for large codes.
  - Therefore, real-world codes are designed with **efficient decoding algorithms**, allowing fast identification of the most likely transmitted codeword.

Imagine that we want to design a code with m message bits and r check bits that will allow all single errors to be corrected. Each of the 2^m legal messages has n illegal codewords at a distance of 1 from it. These are formed by systematically inverting each of the n bits in the n-bit codeword formed from it. Thus, each of the 2^m legal messages requires $n + 1$ bit patterns dedicated to it. Since the total number of bit patterns is 2^n, we must have $(n + 1)2^m \leq 2^n$. Using $n = m + r$, this requirement becomes

$$(m + r + 1) \leq 2^r \qquad (3\text{-}1)$$

Given m, this puts a lower limit on the number of check bits needed to correct single errors.

3.2 ERROR DETECTION AND CORRECTION

3.2.1 Error-Correcting Codes

- **Hamming codes achieve the theoretical lower limit** of redundancy needed for single-error correction.
- **Bit positions are numbered starting from 1**, left to right.
- **Positions that are powers of 2** (1, 2, 4, 8, 16, …) are **check bits** (parity bits).
  All other positions (3, 5, 6, 7, 9, 10, 11, …) carry **data bits**.
- Example: **(11,7) Hamming code** → 11 total bits (7 data + 4 check).
- **Each check bit enforces even (or odd) parity** over a specific set of bits.
- To find which check bits cover a data bit at position *k*:
  → **Write k as a sum of powers of 2**.
  Example:
  - 11 = 1 + 2 + 8 → checked by parity bits at positions 1, 2, 8
  - 29 = 1 + 4 + 8 + 16 → checked by parity bits at positions 1, 4, 8, 16
- **Hamming codes have Hamming distance = 3**, meaning:
  ✓ Can **correct single-bit errors**
  ✓ Can **detect (but not correct) double-bit errors**

Decoding Process

- The receiver **recalculates all check bits** using the received codeword.
- The results of these parity checks are called **check results**.

3.2 ERROR DETECTION AND CORRECTION

- If all check results = 0 → **no error**.
  If not, the pattern of check results forms the **error syndrome**.
- **Error syndrome identifies the exact bit position** that is wrong.
  - Example: syndrome = 0101 (binary) = 5
    → Bit position 5 is in error.
- The receiver **flips the erroneous bit**, then **discards the check bits** to recover the original message.

Practical Notes

- Hamming codes are mainly used in:
  - **Error-correcting memory (ECC RAM)**
  - Simple reliable systems
- Most modern networks use **stronger codes** (LDPC, Reed-Solomon, convolutional codes).

3.2.1 Error Correcting Codes

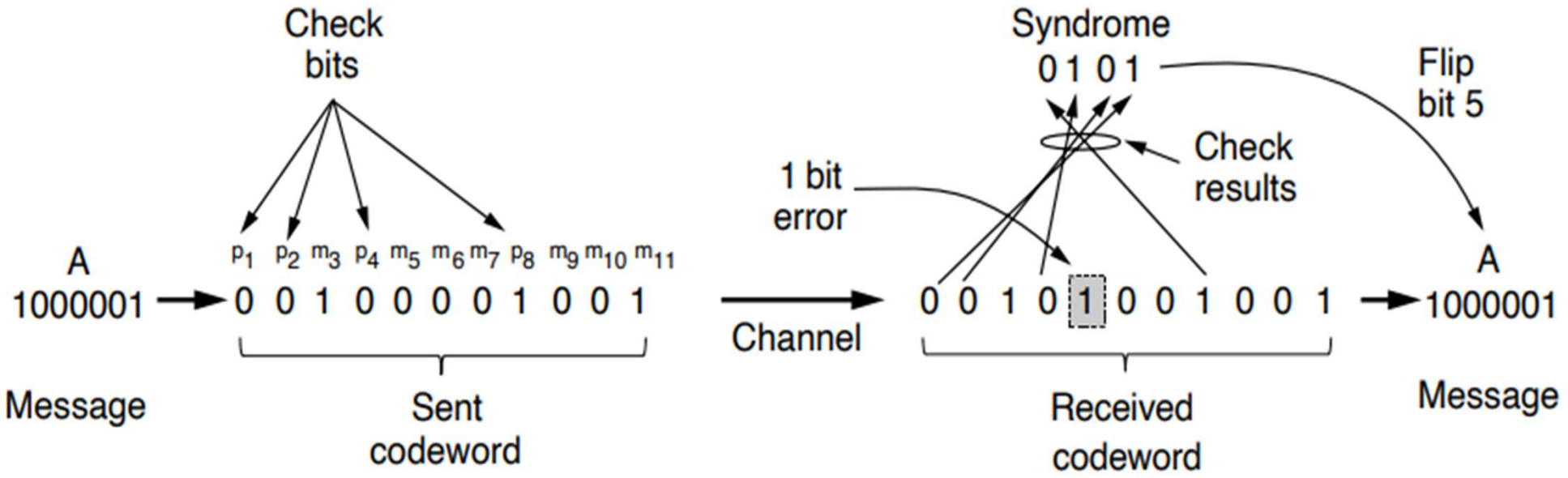

Figure 3-6. Example of an (11, 7) Hamming code correcting a single-bit error.

3.2 ERROR DETECTION AND CORRECTION

Convolutional codes are the second type of error-correcting codes discussed, and **the only one that is not a block code**.

Instead of fixed-size message blocks, the encoder:

- Processes a **continuous stream of input bits**.
- Produces an **output stream** whose values depend on:
  - The **current input bit**.
  - A number of **previous input bits** (memory effect).

The number of previous bits influencing the output is called the **constraint length**.

Convolutional codes are described by:

- **Rate (r)** = (input bits) / (output bits).
- **Constraint length (k)** = number of bits stored in memory that contribute to output.

Widely used in practice, including:

- GSM mobile systems
- Satellite communications
- IEEE 802.11 Wi-Fi
- NASA deep-space missions

Example: NASA (r = 1/2, k = 7) convolutional code (used since Voyager missions):

- For every input bit, **2 output bits** are generated → rate = 1/2.
- Encoder is **binary and linear** (uses XOR operations).
- **Not systematic**: the original input bit does not appear unchanged in the output.

The encoder uses **six internal memory registers**, giving a total constraint length of **7** (current bit + 6 stored bits).

Example of shifting internal state:

- Input sequence **111** with initial state 000000:
  - After input 1 → state: 100000 → output: **11**
  - After input 1 → state: 110000 → output: **10**
  - After input 1 → state: 111000 → output: **01**

It takes **7 shifts** to "flush" a bit completely from the encoder's memory.

Decoding is performed using the **Viterbi algorithm**:

- Searches for the **most likely input bit sequence** that could produce the observed output (including errors).
- For each time step and each possible state, it keeps the best (minimum error) path.
- The path with the fewest error assumptions at the end is chosen as the decoded message.

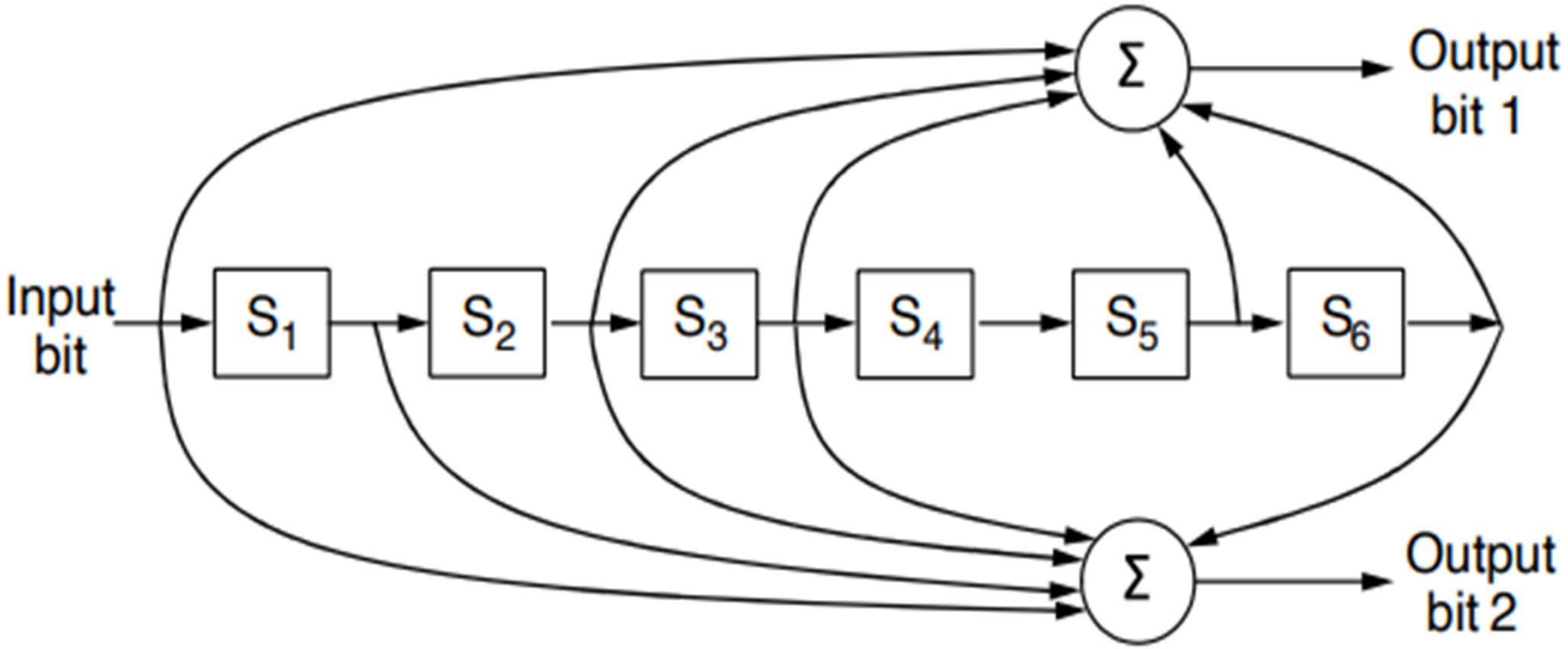

Figure 3-7. The NASA binary convolutional code used in 802.11.

3.2 ERROR DETECTION AND CORRECTION

3.2.1 Error-Correcting Codes

- **Reed–Solomon (RS) codes** are a major class of **block codes** widely used in practice.
- Similar to Hamming codes, RS codes add **redundancy** but operate on **symbols (multi-bit units)** instead of single bits.

Key Concepts

- **Operate on finite fields** (Galois Fields).
- Use **m-bit symbols**; number of possible symbols = $\mathbf{2^m}$.
- A codeword has **n symbols**, of which **m symbols carry data** and **n − m are redundant**.

Principle Behind RS Codes

- Based on the fact that a **degree-n polynomial is uniquely determined by n+1 points**.
- Data points are treated as points on a polynomial curve.
- **Redundant points are computed** so the receiver can reconstruct the polynomial even if some symbols are corrupted.

Intuition / Example

- If two data points define a line, sending:
  - **Two original data points**, plus
  - **Two extra points that lie on the same line**

    gives redundancy.

3.2 ERROR DETECTION AND CORRECTION

Structure

- RS codes typically use parameters like:
  - **(255, 233)** code (common in storage & networking)
    - 255 total symbols
    - 233 data symbols
    - **22 parity symbols** → allow correcting multiple symbol errors.

Decoding

- Uses **Berlekamp–Massey algorithm** for polynomial reconstruction.
- Efficient for medium-length RS codes.
- Handles both:
  - **Random errors**, and
  - **Burst errors** — since entire corrupted symbols can be corrected.

Applications

- Used widely due to strong burst-error protection:
  - CDs, DVDs, Blu-ray
  - QR codes
  - Satellite communications
  - DSL and cable modem systems

3.2 ERROR DETECTION AND CORRECTION

Reed–Solomon (RS) Codes

- RS codes are **strong error-correcting codes**, especially effective for **burst errors**.
- They operate on **symbols of m bits** (not individual bits), so:
  - A **single-bit error** = 1 symbol error
  - A **burst of m bits** = still 1 symbol error
- Widely used in **DSL, cable data, satellites, CDs, DVDs, Blu-ray discs**.
- If a code adds **2t redundant symbols**, it can:
  - **Correct t symbol errors**, regardless of whether errors are bursty or isolated.
- Example:
  - Code (255, 233) → 32 redundant symbols → can correct **16 symbol errors**.
  - Since each symbol is 8 bits → can correct **up to 128 consecutive bit errors**.
- In the case of **erasures** (locations of errors known), an RS code can correct **2t erasures** — twice as many as random errors.
- RS codes are often **combined with convolutional codes**:
  - Convolutional codes correct isolated bit errors.
  - RS codes correct remaining **burst errors** when convolutional decoding fails.
  - Combined system gives strong protection against both types of errors.

LDPC (Low-Density Parity Check) Codes

- LDPC codes were invented by **Gallager (1962)** but only became practical after ~1995 due to computing advancements.
- They are **linear block codes**, represented using sparse (low-density) parity-check matrices.
- Each output bit depends on **only a small subset** of input bits (low number of 1s in the matrix).
- Decoding uses an **iterative approximation algorithm**, which gradually improves the match between the received word and a valid codeword.
- LDPC codes are highly effective for **large block sizes**.
- They provide **excellent error-correction performance**, often exceeding older codes (Hamming, convolutional, and even Reed–Solomon in many cases).
- Adoption is growing rapidly; part of major modern standards:
  - **Digital video broadcasting (DVB-S2)**
  - **10 Gbps Ethernet**
  - **Power-line communication networks**
  - **Latest Wi-Fi (802.11) standards**

3.2 ERROR DETECTION AND CORRECTION

Burst Error Detection Using Parity + Interleaving

Parity limitations

- Single parity bit detects only **odd-numbered bit errors**.
- Bad for long burst errors → error detection probability = 0.5.

Row-wise parity

- Arrange data as a **matrix of k rows × n columns**.
- Compute parity for each row → detect up to **one error per row**.

Interleaving

- Rearranges transmission order to spread burst errors across different rows.
- Turns burst errors into isolated errors that parity bits can detect.
- Example (n = 7, k = 7):
    - Burst of length 7 → after interleaving, becomes at most **1 error per row**.
    - Parity bits in each row detect it.

Limits

3.2.2 Error Detecting Codes

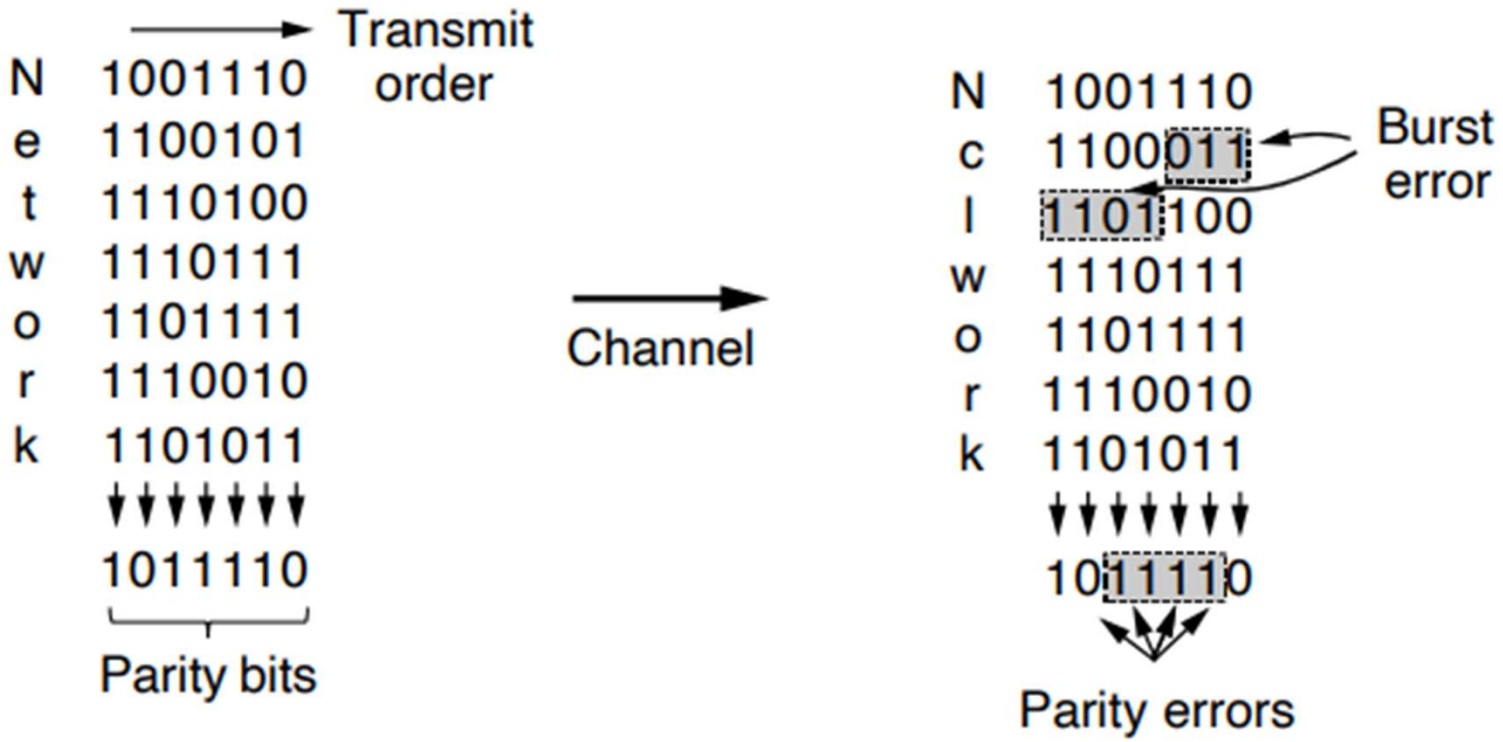

Figure 3-8. Interleaving of parity bits to detect a burst error.

3.2 ERROR DETECTION AND CORRECTION

A **checksum** is an error-detecting code, similar to parity bits, but typically stronger.
It is formed by computing a **sum of groups of bits** (often 16-bit words) and taking the **complement** of that sum.
The checksum is appended to the end of a message; during verification, both data and checksum are summed.

- **If result = 0**, no error is detected.

The widely used **16-bit Internet checksum** (in IP packets) divides the message into 16-bit words and sums them.
This checksum detects errors in *words*, not in single bits; some bit flips that preserve parity may still change the checksum, enabling detection.
The Internet checksum uses **one's-complement arithmetic**, not modulo 2^{16}:

- Negative numbers are represented as the bitwise complement of positive numbers.
- Overflowed high-order bits are wrapped around and added into low-order bits (end-around carry).

One's complement has **two representations of zero** (all 0s and all 1s), allowing a checksum field to be all zeros to indicate "no checksum" if needed.
Compared to two's complement, one's complement offers **better uniformity** and ensures no information is lost due to overflow.
Traditional checksum algorithms assume that **frame data is random**, but real-world data is not random (Partridge et al., 1995).
Because of this incorrect assumption, **undetected errors occur more often** than previously believed.
The **Internet checksum** is simple and efficient but has limitations:

- Fails to detect **deletion or addition of zero-valued data**.
- Cannot detect **swapping of data blocks** inside the message.
- Provides **weak protection against message splicing**, where parts of two packets are combined.

3.2 ERROR DETECTION AND CORRECTION

These types of errors are unlikely in random data but **common in faulty/buggy hardware**.
Fletcher's checksum (Fletcher, 1982) is a better alternative:

- Adds a **positional component** by multiplying data by its position index.
- Stronger at detecting **reordering or positional changes** in the data.

At the link layer, an even stronger method is widely used: **Cyclic Redundancy Check (CRC)**.
CRC (polynomial code):

- Treats bit strings as **polynomials with binary coefficients (0 or 1)**.
- Provides **much stronger error detection** than simple checksums or Fletcher's checksum.

regarded as the coefficient list for a polynomial with k terms, ranging from x^{k-1} to x^0. Such a polynomial is said to be of degree $k-1$. The high-order (leftmost) bit is the coefficient of x^{k-1}, the next bit is the coefficient of x^{k-2}, and so on. For example, 110001 has 6 bits and thus represents a six-term polynomial with coefficients 1, 1, 0, 0, 0, and 1: $1x^5 + 1x^4 + 0x^3 + 0x^2 + 0x^1 + 1x^0$.

Polynomial arithmetic is done modulo 2, according to the rules of algebraic field theory. It does not have carries for addition or borrows for subtraction. Both addition and subtraction are identical to exclusive OR. For example:

10011011	00110011	11110000	01010101
+ 11001010	+ 11001101	− 10100110	− 10101111
01010001	11111110	01010110	11111010

3.2 ERROR DETECTION AND CORRECTION

CRC uses **long-division in binary**, but subtraction is performed **modulo 2**.
A **divisor** (generator polynomial *G(x)*) is chosen so that it “goes into” the dividend with the same number of bits.
Sender and receiver must **agree on the same generator polynomial** beforehand.
To compute the CRC:

1. Let *r* be the degree of *G(x)* and append **r zero bits** to the message polynomial *M(x)*.
2. **Divide the extended message** by *G(x)* using modulo-2 binary division.
3. **Remainder** becomes the CRC bits (checksum). Append to form transmitted frame *T(x)*.

If the receiver divides the received frame by *G(x)* and the **remainder is not zero**, a transmission error occurred.
Example shows dividing a bitstring (11001101111) by generator $G(x) = x^4 + x + 1$.
Highlights that after subtracting the remainder from the dividend, the remainder of the result modulo *G(x)* is zero (a property of division).
Explains a similar decimal analogy: dividing 210,278 by 10,941 leaves remainder 2,399; subtracting the remainder gives a number divisible by 10,941.
Discusses **transmission errors**: if received frame is *T(x)* + *E(x)*, the bits in *E(x)* indicate which bits were flipped.
A **single-bit error** corresponds to an *E(x)* with a 1 at the error position (surrounded by zeros).

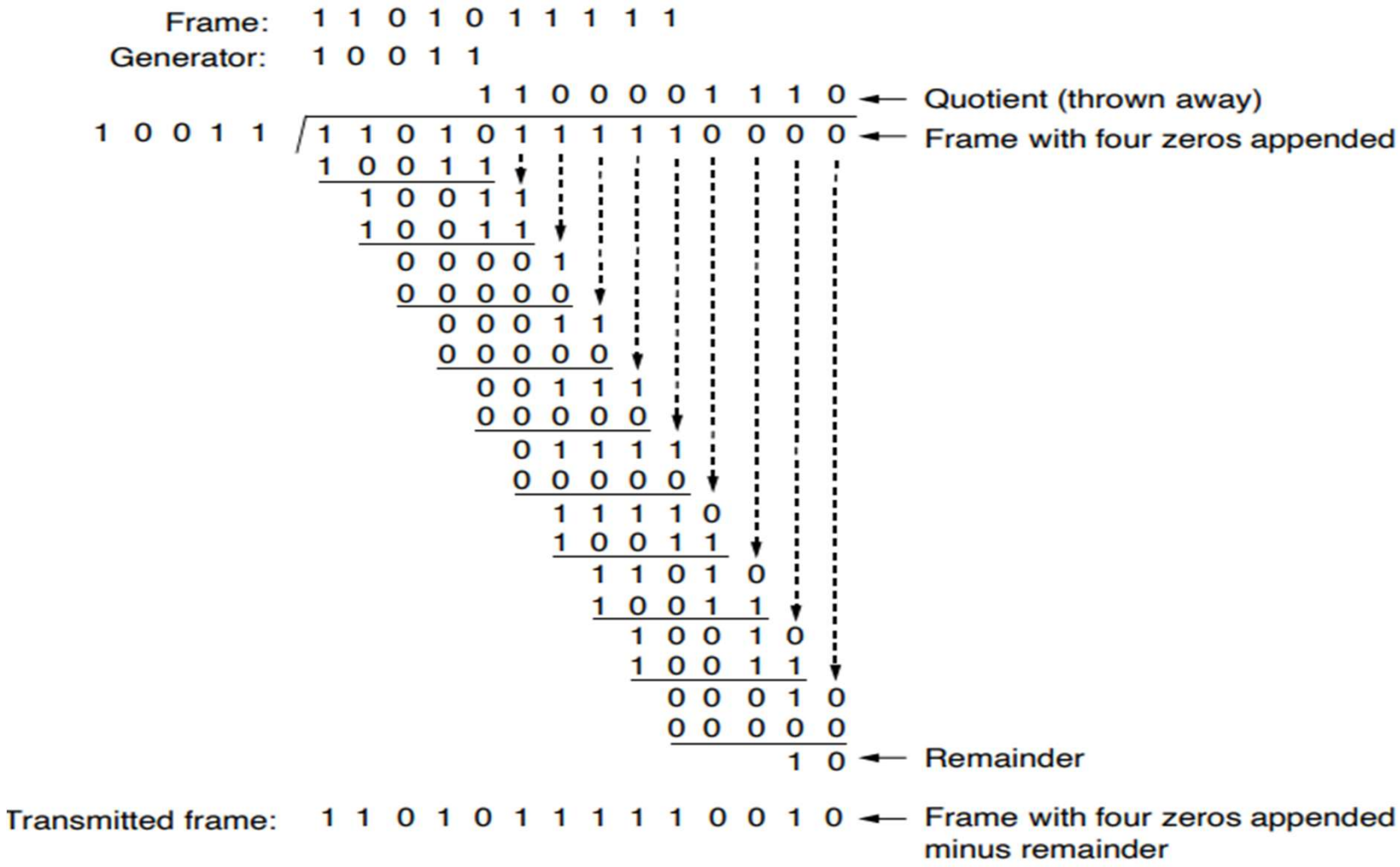

Figure 3-9. Example calculation of the CRC.

3.2 ERROR DETECTION AND CORRECTION

- When the receiver gets the checksummed frame, it divides it by the generator polynomial **G(x)**.
- The receiver computes **T(x) + E(x)** divided by G(x). Since **T(x)/G(x) = 0**, the result reduces to **E(x)/G(x)**.
- Any error pattern **E(x)** divisible by **G(x)** will NOT be detected; all others will be.
- **Single-bit errors**:
  - A 1-bit error yields $\mathbf{E(x) = x^i}$.
  - A good generator **G(x)** must contain **at least two terms**, ensuring it cannot divide $\mathbf{x^i}$, so all single-bit errors are detected.
- **Double-bit errors**:
  - Two isolated 1-bit errors produce $\mathbf{E(x) = x^i + x^j = x^j(x^{i-j} + 1)}$.
  - To detect all double-bit errors, choose **G(x)** that does **not** divide $\mathbf{x^k + 1}$ for any k within the frame length.
  - Low-degree generator polynomials such as $\mathbf{x^{15} + x^{14} + 1}$ do not divide $\mathbf{x^k + 1}$ for k < 32,768.
- **Odd number of bit errors**:
    - If the number of flipped bits is odd, **E(x)** has an odd number of terms.
    - No polynomial with an odd number of terms can divide **x + 1** in modulo-2 arithmetic.
  - By ensuring **G(x)** includes **x + 1** as a factor, all odd-numbered bit errors are detected.
- **Burst errors**:
  - A burst of length ≤ r (where r = number of CRC check bits) is represented as:
    $\mathbf{E(x) = x^i (x^{k+1} + ... + 1)}$
  - If **G(x)** includes a constant term ($\mathbf{x^0}$), it cannot divide $\mathbf{x^k}$, so burst errors shorter than or equal to bits are always detected.
- A well-chosen generator polynomial can detect:
    - All single-bit errors
    - All double-bit errors (within limits)
    - All errors with an odd number of bad bits
    - All burst errors of length ≤ r

3.2 ERROR DETECTION AND CORRECTION

Certain polynomials have become international standards. The one used in IEEE 802 followed the example of Ethernet and is

$$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} \text{ ½}^{r-1}\text{.} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x^1 + 1$$

3.2.2 Error-Detecting Codes (CRC)

- A burst error of length **r + 1** will produce a zero remainder (go undetected) **only if the burst pattern exactly matches G(x)**.
- Because a burst has its **first and last bits = 1**, only the **r – 1 internal bits** determine whether it equals G(x).
- If all bit patterns are equally likely, the probability that a burst of length **r + 1** goes undetected is **1 / 2^(r–1)**.
- A commonly used CRC polynomial detects:
  - **All burst errors of length ≤ 32**
  - **All errors that flip an odd number of bits**
- This polynomial has been widely used since the 1980s but is **not optimal** by modern standards.
- Castagnoli et al. (1993) and Koopman (2002) identified **better CRC polynomials** through exhaustive search.
- These improved CRCs achieve a **Hamming distance of 6** for typical message sizes, compared to **distance 4** for the standard **IEEE CRC-32**.
- Despite appearing mathematically complex, CRCs are easy to implement in hardware using **shift-register circuits**.
- Hardware computation of CRCs is standard in networking equipment and is used in:
  - **LAN technologies** (Ethernet, IEEE 802.11 Wi-Fi)
  - **Point-to-point links** (e.g., SONET)

3.3 ELEMENTARY DATA LINK PROTOCOLS

3.3.1 Initial Simplifying Assumptions

Independent Processes

- The **physical layer**, **data link layer**, and **network layer** are treated as **independent processes** communicating by passing messages.
- A common implementation:
  - Physical layer + part of data link layer → run on **NIC hardware**.
  - Rest of data link layer + network layer → run on **main CPU**, often as the **device driver** plus OS code.
- Other implementations exist:
  - All three layers offloaded to a **network accelerator**.
  - All layers on the **main CPU** in a **software-defined radio**.
- Preferred implementations change over time as hardware/technology evolve.
- Treating layers as separate processes:
  - Makes discussion **cleaner**.
  - Reinforces **layer independence**.

Unidirectional Communication

- Assume **machine A sends a long stream of data to machine B**.
- The communication is **reliable and connection-oriented**.
- Only A → B is considered first; bidirectional traffic will be handled later.
- Assume A has **infinite data ready**, meaning:
  - A never waits for data to be generated.
  - Its network layer always responds instantly to send requests.
- This restriction is later removed.

Reliable Machines and Processes

- Machines are assumed **not to crash**; no failures or reboots.
- Protocols focus on **communication errors**, not system failures.
- For the data link layer:
  - Packets from the network layer are treated as **pure data bits**.
  - Interpretation of headers by the destination network layer is **irrelevant** to the link layer.

3.3 Elementary Data Link Protocols
3.3.1 Initial Simplifying Algorithms

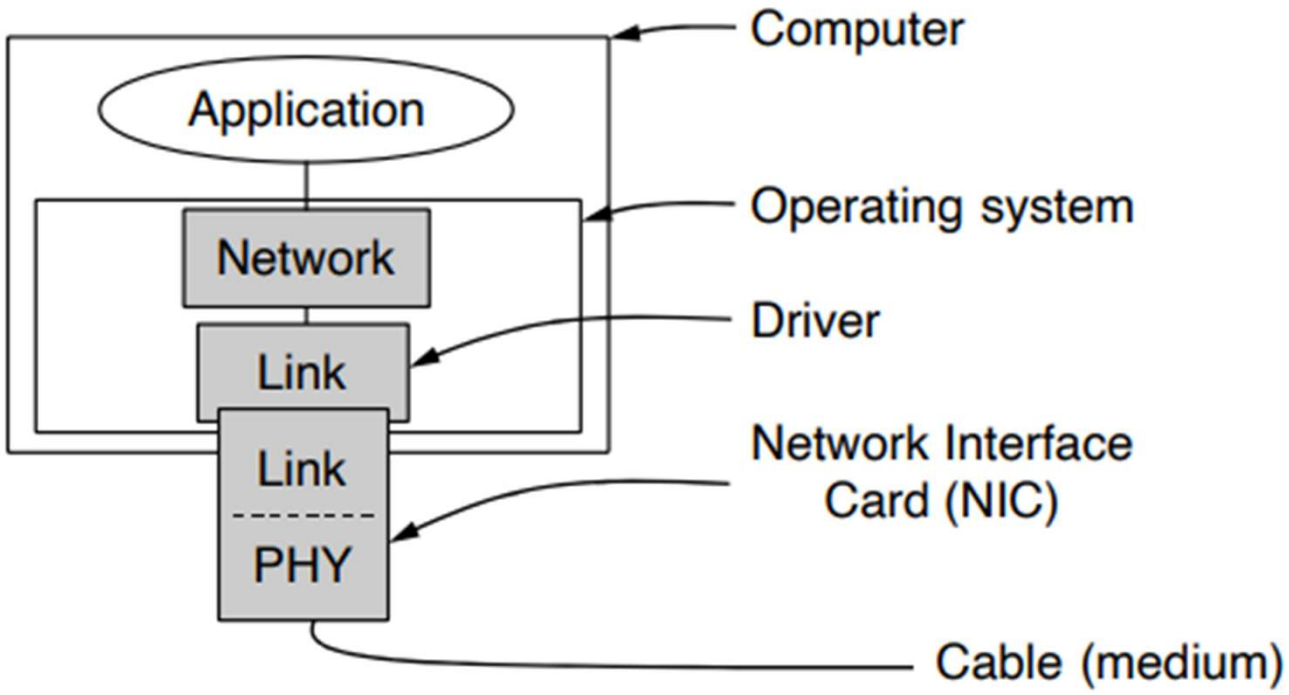

Figure 3-10. Implementation of the physical, data link, and network layers.

3.3 ELEMENTARY DATA LINK PROTOCOLS

3.3.2 Basic Transmission and Receipt

- When the data link layer receives a packet from the network layer, it **encapsulates** it into a frame by adding:
  - a **header** (control information)
  - a **trailer** (typically containing a checksum)
- Frames are sent to the receiver's data link layer using library procedures that:
  - handle sending and receiving frames through the physical layer
  - compute, append, and verify checksums (often done in hardware)
  - hide checksum details from the protocol description
- The receiver initially does nothing but **wait for an event**.
  - This is modeled by `wait_for_event(&event)`
  - The function returns only when something happens (e.g., frame arrival, checksum error)
- In real systems, event handling is usually done by **interrupts**, not waiting loops, but this detail is ignored for conceptual simplicity.
- When a frame arrives:
  - The receiver recomputes the **checksum**
    - If incorrect → event = **cksum_err**
    - If correct → event = **frame_arrival**
  - On a good frame, the receiver extracts the **packet portion** and passes it to the network layer.
  - **The frame header is never given to the network layer.**
- Reason for this restriction:
  - Ensures **clean separation** between data link and network layer protocols.
  - Allows the frame format or data link behavior to change (e.g., new NIC installation) **without modifying network layer software**.
  - This rigid interface simplifies system design and allows independent evolution of protocol layers.

3.3 ELEMENTARY DATA LINK PROTOCOLS

3.3.2 Basic Transmission and Receipt (Data Structures & Interfaces)

Defined Data Structures

- **Boolean**
  - Enumerated type: `true` or `false`.
- **seq_nr**
  - Small integer used as a **sequence number** for frames.
  - Ranges from **0 to MAX_SEQ**, defined per protocol.
- **packet**
  - Unit of information exchanged:
    - Between **network layer ↔ data link layer** (same machine)
    - Between **peer network layers** on different machines
  - In the simplified model: fixed size **MAX_PKT bytes** (real systems use variable length).
- **frame_kind**
  - Enumerated type describing the **kind of frame** (e.g., data, ack, etc.).
- **frame**
  - Contains **four fields**:
    - `kind` – specifies if the frame carries data or just control info
    - `seq` – sequence number
    - `ack` – acknowledgment number
    - `info` – contains a **packet** when it is a data frame
  - First three fields form the **frame header**.

Relationship Between Packet and Frame

- **Network layer** builds packets by:
  - Taking a transport-layer message
  - Adding a **network-layer header**
- Packets are then passed to the **data link layer** to be placed in the frame's **info** field.
- At the destination:
  - Data link layer removes the packet from the frame
  - Passes it to the **network layer**

Library Procedures (from Fig. 3-11)

- **wait_for_event(&event)**
  - Blocks until something happens (e.g., frame arrival, checksum error).
- **to_network_layer(pkt)**
  - Data link layer → Network layer (send packet up).
- **from_network_layer(pkt)**
  - Network layer → Data link layer (get packet to send).
- **to_physical_layer(frame)**
  - Data link layer → Physical layer (send frame out).
- **from_physical_layer(frame)**
  - Physical layer → Data link layer (receive frame).

Layer Interface Mapping

- **Layer 2 ↔ Layer 3:** `to_network_layer`, `from_network_layer`
- **Layer 1 ↔ Layer 2:** `to_physical_layer`, `from_physical_layer`

3.3.2 Basic Transmission and Receipt

```
#define MAX_PKT 1024                                        /* determines packet size in bytes */
typedef enum {false, true} boolean;                         /* boolean type */
typedef unsigned int seq_nr;                                /* sequence or ack numbers */
typedef struct {unsigned char data[MAX_PKT];} packet;       /* packet definition */
typedef enum {data, ack, nak} frame_kind;                   /* frame_kind definition */

typedef struct {                                            /* frames are transported in this layer */
  frame_kind kind;                                          /* what kind of frame is it? */
  seq_nr seq;                                               /* sequence number */
  seq_nr ack;                                               /* acknowledgement number */
  packet info;                                              /* the network layer packet */
} frame;
/* Wait for an event to happen; return its type in event. */
void wait_for_event(event_type *event);

/* Fetch a packet from the network layer for transmission on the channel. */
void from_network_layer(packet *p);

/* Deliver information from an inbound frame to the network layer. */
void to_network_layer(packet *p);

/* Go get an inbound frame from the physical layer and copy it to r. */
void from_physical_layer(frame *r);

/* Pass the frame to the physical layer for transmission. */
void to_physical_layer(frame *s);
```

```
/* Start the clock running and enable the timeout event. */
void start_timer(seq_nr k);

/* Stop the clock and disable the timeout event. */
void stop_timer(seq_nr k);

/* Start an auxiliary timer and enable the ack_timeout event. */
void start_ack_timer(void);

/* Stop the auxiliary timer and disable the ack_timeout event. */
void stop_ack_timer(void);

/* Allow the network layer to cause a network_layer_ready event. */
void enable_network_layer(void);

/* Forbid the network layer from causing a network_layer_ready event. */
void disable_network_layer(void);

/* Macro inc is expanded in-line: increment k circularly. */
#define inc(k) if (k < MAX_SEQ) k = k + 1; else k = 0
```

Figure 3-11. Some definitions needed in the protocols to follow. These definitions are located in the file *protocol.h*.

3.3 ELEMENTARY DATA LINK PROTOCOLS

3.3.2 Basic Transmission and Receipt (Timers, Flow Control & Sequence Numbers)

Unreliable Channel & Timers

- The channel may **lose entire frames**, so the sender must use a **timer** whenever it transmits a frame.
- If no acknowledgement or reply arrives before the timer expires:
  - A **timeout event** occurs → `event = timeout`
- **start_timer** and **stop_timer**:
  - `start_timer` begins or **resets** the timer (even if already running).
  - `stop_timer` stops it
- Timeout events only occur while the timer is running.

ACK Timer

- **start_ack_timer** and **stop_ack_timer** control a second timer used to trigger **acknowledgement frames** when needed.
- Used in more sophisticated protocols that delay or batch acknowledgements.

Network Layer Control

- **enable_network_layer** allows the network layer to notify the data link layer when it has a packet ready (`event = network_layer_ready`).
- **disable_network_layer** prevents such notifications.
- Proper enabling/disabling avoids the network layer **overloading** the data link layer with packets when buffers are full.

3.3 ELEMENTARY DATA LINK PROTOCOLS

Sequence Numbers

- Sequence numbers range from **0 to MAX_SEQ**, determined per protocol.
- Often need to advance the sequence number **circularly**.
    - The macro **inc** handles circular increment (e.g., MAX_SEQ → 0).

Why use a macro?

- Performance is often limited by **protocol processing**, not transmission speed.
- Using macros for small, frequent operations like incrementing:
    - Keeps code readable
    - Improves performance by in-lining

Protocol Declarations

- The definitions in Fig. 3-11 apply to all protocols.
- In actual C code:
    - These definitions are stored in **protocol.h**
    - Protocol files include them via `#include "protocol.h"`

3.3 ELEMENTARY DATA LINK PROTOCOLS

3.3.3 Simplex Link-Layer Protocols (Utopia Protocol)

Environment & Assumptions

- Data flows in **one direction only** (simplex).
- Network layers at both sender and receiver are **always ready**.
- **No errors**, **no frame loss**, and **no damage** ever occurs on the channel.
- **Infinite buffer space** is available.
- **No delays**—processing time is ignored.
- This protocol is intentionally **unrealistic**, used only to illustrate basic structure.

Protocol Characteristics

- Named **"Utopia"** because nothing can go wrong.
- No **sequence numbers**, **acknowledgements**, or **flow control**.
    - Thus, **MAX_SEQ** is not used.
- Only possible event: **frame_arrival**.

Sender Behavior

- Runs in the **data link layer** of the source machine.
- Executes an **infinite loop**:
    - Fetch a packet from the always-ready network layer.
    - Construct a frame using the frame variable `s`.
    - Send the frame to the physical layer.
- Uses **only the info field** of the frame.
    - No control fields are needed because no errors or flow control exist.

Receiver Behavior

- Runs in the **data link layer** of the destination machine.
- Initially waits for an event → always **frame_arrival**.
- When a frame arrives:
    1. `from_physical_layer(r)` picks up the received frame.
    2. The **data** in `r.info` is delivered to the network layer.
    3. The receiver waits again for the next frame.

3.3 ELEMENTARY DATA LINK PROTOCOLS

Nature of the Protocol

- Does **not** handle:
  - Flow control
  - Error detection
  - Error correction

- Roughly resembles an **unacknowledged connectionless service**, though even that would still include **basic error detection**.

3.3.3 Simplex Link Layer Protocols
Utopia: No Flow Control or Error Correction

```
/* Protocol 1 (Utopia) provides for data transmission in one direction only, from
   sender to receiver. The communication channel is assumed to be error free
   and the receiver is assumed to be able to process all the input infinitely quickly.
   Consequently, the sender just sits in a loop pumping data out onto the line as
   fast as it can. */
typedef enum {frame_arrival} event_type;
#include "protocol.h"
void sender1(void)
{
  frame s;                                  /* buffer for an outbound frame */
  packet buffer;                            /* buffer for an outbound packet */
  while (true) {
      from_network_layer(&buffer);          /* go get something to send */
      s.info = buffer;                      /* copy it into s for transmission */
      to_physical_layer(&s);                /* send it on its way */
  }                                         /* Tomorrow, and tomorrow, and tomorrow,
                                               Creeps in this petty pace from day to day
                                               To the last syllable of recorded time.
                                                   — Macbeth, V, v */
}
void receiver1(void)
{
  frame r;
  event_type event;                         /* filled in by wait, but not used here */
  while (true) {
      wait_for_event(&event);               /* only possibility is frame_arrival */
      from_physical_layer(&r);              /* go get the inbound frame */
      to_network_layer(&r.info);            /* pass the data to the network layer */
  }
}
```

Figure 3-12. A utopian simplex protocol.

3.3 ELEMENTARY DATA LINK PROTOCOLS

The Problem

- Sender may transmit frames **faster than the receiver can process them**.
- Without protection, the receiver could be overwhelmed (buffer overflow, dropped frames).
- Channel is still assumed **error-free**, and traffic is **simplex** (data only from sender → receiver).

Inefficient "Brute Force" Solution

- Build a receiver powerful enough to handle a continuous back-to-back stream at line rate.
- Requires large buffering + high processing speed.
- Often wasteful because link utilization is usually variable/low.
- Simply moves the overload problem to the **network layer**.

General Solution: Feedback

- Receiver sends back a **dummy frame** after delivering a packet to its network layer.
- This dummy frame acts as **permission** for the sender to transmit the next data frame.
- This feedback frame is an **acknowledgement (ACK)**.

Stop-and-Wait Protocol

- Sender transmits **one frame**, then **waits** for an acknowledgement.
- After ACK is received, sender sends the next frame.
- Ensures the receiver never gets more than one unprocessed frame at a time.

Communication Characteristics

- Data flow is simplex (sender → receiver).
- But acknowledgements require frames in **both directions**.
- Therefore the physical channel must support **bidirectional communication**.
- A **half-duplex** channel is sufficient due to strict alternation:
    - Sender sends frame → Receiver sends ACK → Sender sends next frame → …

3.3 ELEMENTARY DATA LINK PROTOCOLS

Sender Behavior (Differences from Utopia Protocol)

- After sending a frame, the sender **must pause and wait** for the ACK.
- Only one type of incoming frame exists: the ACK.
- Sender does **not need to inspect** the ACK content.

Receiver Behavior

- After receiving and delivering a data frame to the network layer:
    - Receiver **immediately sends an acknowledgement frame** back to the sender.
- ACK frame content is irrelevant—it only signals arrival.

Purpose

- Implements **simple flow control**.
- Ensures sender never floods receiver.
- Builds the foundation for more sophisticated link-layer protocols.

Adding Error Correction: Sequence Numbers and ARQ

```
/* Protocol 2 (Stop-and-wait) also provides for a one-directional flow of data from
   sender to receiver. The communication channel is once again assumed to be error
   free, as in protocol 1. However, this time the receiver has only a finite buffer
   capacity and a finite processing speed, so the protocol must explicitly prevent
   the sender from flooding the receiver with data faster than it can be handled. */
typedef enum {frame_arrival} event_type;
#include "protocol.h"
void sender2(void)
{
  frame s;                              /* buffer for an outbound frame */
  packet buffer;                        /* buffer for an outbound packet */
  event_type event;                     /* frame_arrival is the only possibility */
  while (true) {
      from_network_layer(&buffer);      /* go get something to send */
      s.info = buffer;                  /* copy it into s for transmission */
      to_physical_layer(&s);            /* bye-bye little frame */
      wait_for_event(&event);           /* do not proceed until given the go ahead */
  }
}
void receiver2(void)
{
  frame r, s;                           /* buffers for frames */
  event_type event;                     /* frame_arrival is the only possibility */
  while (true) {
      wait_for_event(&event);           /* only possibility is frame_arrival */
      from_physical_layer(&r);          /* go get the inbound frame */
      to_network_layer(&r.info);        /* pass the data to the network layer */
      to_physical_layer(&s);            /* send a dummy frame to awaken sender */
  }
}
```

Figure 3-13. A simplex stop-and-wait protocol.

3.3 ELEMENTARY DATA LINK PROTOCOLS

3.3.3 Simplex Link-Layer Protocols Adding Error Correction: Sequence Numbers and ARQ

Channel Behavior and Assumptions

- Frames can be **lost** or **damaged**.
- Damaged frames are detected by the **checksum** at the receiver.
- If a frame is damaged in a way that still produces a valid checksum, the protocol can fail (this is true for all link-layer protocols).

Initial Idea (Flawed Approach)

- Modify stop-and-wait to include a **timer**:
  - Sender sends a frame.
  - Receiver sends an **ACK** only if the frame arrives undamaged.
  - Damaged frames are discarded.
  - If sender's timer expires, it **retransmits** the frame.
- This seems workable but contains a **serious flaw**.

Requirement of Data Link Layer

- Must deliver to the receiver's network layer a **perfect, duplicate-free** sequence of packets.
- Network layer cannot detect:
  - **Lost packets**
  - **Duplicate packets**
- Therefore, the data link layer must ensure that **no combination of errors** leads to duplicates being delivered.

Failure Scenario (Why the naive scheme is broken)

1. A sends **packet 1**.
   - B receives it correctly and passes it to its network layer.
   - B sends an ACK back to A.
2. The ACK is **lost**.
3. A **times out**, assumes the data frame was lost, and retransmits **packet 1**.
4. B receives the **duplicate frame** intact and **passes it again** to its network layer.
   - Result: **Duplicate packet delivered**, corrupting the data stream (e.g., duplicated part of a file).

3.3 ELEMENTARY DATA LINK PROTOCOLS

3.3.3 Simplex Link-Layer Protocols Adding Error Correction: Sequence Numbers and ARQ

The receiver must distinguish **new frames** from **retransmissions** to avoid delivering duplicates.

The solution: add a **sequence number** to each transmitted frame.

Sequence number space must be large enough for the protocol to function correctly; smaller numbers mean more efficiency but risk ambiguity.

For this protocol, the only possible ambiguity is between:

- **Frame m** (current frame), and
- **Frame m + 1** (the next frame)

If frame *m* is lost/damaged, the receiver will not send an ACK → sender retransmits *m* until ACK arrives.

Once *m* is correctly received:

- Receiver sends ACK for *m*
- Sender may receive ACK or may lose it → sender may resend *m* or move to *m* + *1* depending on ACK arrival.

Importantly:

- Sender transmits frame *m* + *1* **only if** ACK for *m* arrived, meaning *m* was definitely received.
- Thus, ambiguity never involves frames beyond immediate neighbors (never between *m* – *1* and *m* + *1*).

Therefore, only **two sequence numbers** are needed → a **1-bit sequence number** (0 or 1).

Receiver always expects one specific sequence number:

- If correct seq number → accept, deliver to network layer, send ACK.
- Then receiver flips expected seq number (modulo 2).

If wrong seq number → frame is a **duplicate**, so receiver discards it but **repeats the last ACK**.

- This ensures sender eventually learns the frame has been received.

3.3 ELEMENTARY DATA LINK PROTOCOLS

3.3.3 Simplex Link-Layer Protocols Adding Error Correction: Sequence Numbers and ARQ

Protocol shown in Fig. 3-14 is an example of an ARQ (Automatic Repeat reQuest) or PAR (Positive Acknowledgement with Retransmission) protocol.

Like earlier protocols, data still flows **only in one direction**, but acknowledgements travel in the opposite direction.

Both sender and receiver maintain **state variables**:

- **next_frame_to_send** (sender): sequence number of the next frame to transmit.
- **frame_expected** (receiver): sequence number of the next valid frame the receiver expects.

Before entering the infinite loop, both sides perform a short initialization phase.

After sending a frame, the sender **starts a timer** (or resets it if already running).

The timeout interval must allow enough time for:

- the frame to travel to the receiver,
- receiver processing,
- and the acknowledgment to return.

If the timeout interval is too short → unnecessary retransmissions occur (hurts performance but not correctness).

After sending a frame, the sender waits for one of three events:

- **A valid (undamaged) ACK arrives**
- **A damaged ACK arrives**
- **The timer expires**

If a valid ACK arrives:

- Sender fetches the next packet from its network layer.
- Overwrites the old packet in the buffer.
- Advances the sequence number.

If a damaged ACK arrives **or** the timer expires:

- Sender keeps both the buffer contents and the sequence number unchanged.
- This ensures it can retransmit the same frame (duplicate).

3.3 ELEMENTARY DATA LINK PROTOCOLS

In all situations, the sender then transmits the buffer contents (either a new frame or a duplicate).
Receiver behavior:

- When a **valid frame** arrives, its sequence number is checked.
- If it is **not a duplicate**, the frame is:
  - accepted,
  - delivered to the network layer,
  - and acknowledged.
- If it is a **duplicate** or **damaged**, the frame is discarded.
- However, the receiver still sends an ACK for the **last correctly received frame**.

This ACK repetition helps the sender determine whether to send the next frame or retransmit a missing one.

3.4 Improving Efficiency
3.4.1 Goal: Bidirectional Transmission, Multiple frames in flight
Bidirectional Transmission: Piggybacking

```
/* Protocol 3 (PAR) allows unidirectional data flow over an unreliable channel. */

#define MAX_SEQ 1                                   /* must be 1 for protocol 3 */
typedef enum {frame_arrival, cksum_err, timeout} event_type;
#include "protocol.h"

void sender3(void)
{
  seq_nr next_frame_to_send;                        /* seq number of next outgoing frame */
  frame s;                                          /* scratch variable */
  packet buffer;                                    /* buffer for an outbound packet */
  event_type event;

  next_frame_to_send = 0;                           /* initialize outbound sequence numbers */
  from_network_layer(&buffer);                      /* fetch first packet */
  while (true) {
      s.info = buffer;                              /* construct a frame for transmission */
      s.seq = next_frame_to_send;                   /* insert sequence number in frame */
      to_physical_layer(&s);                        /* send it on its way */
      start_timer(s.seq);                           /* if answer takes too long, time out */
      wait_for_event(&event);                       /* frame_arrival, cksum_err, timeout */
      if (event == frame_arrival) {
          from_physical_layer(&s);                  /* get the acknowledgement */
          if (s.ack == next_frame_to_send) {
              stop_timer(s.ack);                    /* turn the timer off */
              from_network_layer(&buffer);          /* get the next one to send */
              inc(next_frame_to_send);              /* invert next_frame_to_send */
          }
      }
  }
}
```

```
void receiver3(void)
{
  seq_nr frame_expected;
  frame r, s;
  event_type event;

  frame_expected = 0;
  while (true) {
       wait_for_event(&event);                  /* possibilities: frame_arrival, cksum_err */
       if (event == frame_arrival) {            /* a valid frame has arrived */
            from_physical_layer(&r);            /* go get the newly arrived frame */
            if (r.seq == frame_expected) {      /* this is what we have been waiting for */
                 to_network_layer(&r.info);     /* pass the data to the network layer */
                 inc(frame_expected);           /* next time expect the other sequence nr */
            }
            s.ack = 1 – frame_expected;         /* tell which frame is being acked */
            to_physical_layer(&s);              /* send acknowledgement */
       }
  }
}
```

Figure 3-14. A positive acknowledgement with retransmission protocol.

3.4 IMPROVING EFFICIENCY

3.4.1 Goal: Bidirectional Transmission, Multiple Frames in Flight

Need for Better Link-Layer Efficiency

- Earlier protocols handled **simplex communication** (data in only one direction).
- Real systems typically require **bidirectional data transfer**.
- Link layer performance can improve further if multiple frames can be in flight without waiting for individual acknowledgements.

Piggybacking for Bidirectional Transmission

- A naive way to achieve bidirectional communication:
  Run **two independent simplex protocols**, one in each direction.
- This wastes reverse-channel capacity because acknowledgement frames are tiny compared to data frames.
- A better design uses **one shared link** for data flowing in both directions.
- Data frames and acknowledgements are **intermixed** on the same channel.
- The **kind** field in the frame header distinguishes data frames from acknowledgement frames.

Piggybacking:

  - When a receiver gets a data frame, it **does not immediately send an ACK**.
  - Instead, it waits until it has a data packet to send back.
  - The ACK is attached to (piggybacked on) the next outgoing data frame using the **ack** field in the header.
  - The ACK "gets a free ride," reducing overhead and improving efficiency.

Motivation for Sliding Window (introduced next)

- To improve throughput even further, protocols can allow the sender to have **multiple outstanding frames** before getting acknowledgements.
- This concept—**sliding window**—enables higher link utilization.

3.4 IMPROVING EFFICIENCY

3.4.1 Goal: Bidirectional Transmission, Multiple Frames in Flight

Advantages of Piggybacking

- Saves channel bandwidth by **avoiding separate ACK frames**.
- The **ack field uses only a few bits** in the header, unlike a standalone frame that needs:
  - a full header
  - the acknowledgement
  - a checksum
- Reduces the **total number of frames sent**, decreasing processing overhead at the receiver.
- In many protocols, the piggyback ACK requires only **1 bit** in the header (rarely more than a few).

Complications Introduced by Piggybacking

- The receiver must decide **how long to wait** for a new outgoing data packet to attach the ACK to.
- Waiting too long may exceed the sender's **timeout interval**, causing unnecessary retransmissions.
- If the data link layer could predict arrivals from the network layer, it could choose optimally—but it cannot.
- Therefore, practical protocols use **ad hoc methods**, typically:
  - Wait a fixed, short time for a packet.
  - If a packet arrives quickly → **piggyback the ACK**.
  - If no packet arrives before the timer expires → **send a standalone ACK**.

3.4 IMPROVING EFFICIENCY

3.4.1 Goal: Bidirectional Transmission, Multiple Frames in Flight

Sliding Windows

- Sliding-window protocols are **bidirectional** link-layer protocols enabling simultaneous sending and receiving of frames.
- These protocols differ in **efficiency**, **complexity**, and **buffer requirements**.
- Each outbound frame carries a **sequence number** ranging from 0 up to some maximum.
    - Maximum is usually 2n−12^n - 12n−1, fitting sequence numbers in an n-bit field.
    - Simple stop-and-wait uses n=1n = 1n=1 → sequence numbers 0 and 1.
    - More advanced sliding-window protocols use larger nnn.
- The **sender maintains a sending window**, representing the set of sequence numbers it is allowed to send at any moment.
    - Frames with sequence numbers inside this window are permitted to be transmitted.
- The **receiver also maintains a receiving window**, indicating which sequence numbers it is willing to accept.
    - Sender and receiver windows **do not need to have the same bounds or size**.
    - Some protocols have fixed window sizes, others grow or shrink dynamically over time
- Despite increased flexibility, the protocol **must still deliver packets to the destination network layer in order**, exactly as they were passed from the source network layer.
- The communication channel is assumed to be **reliable in order** (“wire-like”)—it preserves frame order.
- Within the sending window:
    - Frames may have been **sent but not yet acknowledged**, or are **ready to be sent**.
    - When a new packet arrives from the network layer, it gets the next sequence number, and the **window’s upper edge moves forward**.
- When an acknowledgement is received:
    - The **lower edge** of the window is advanced by one, freeing space for new frames.

Sliding Windows

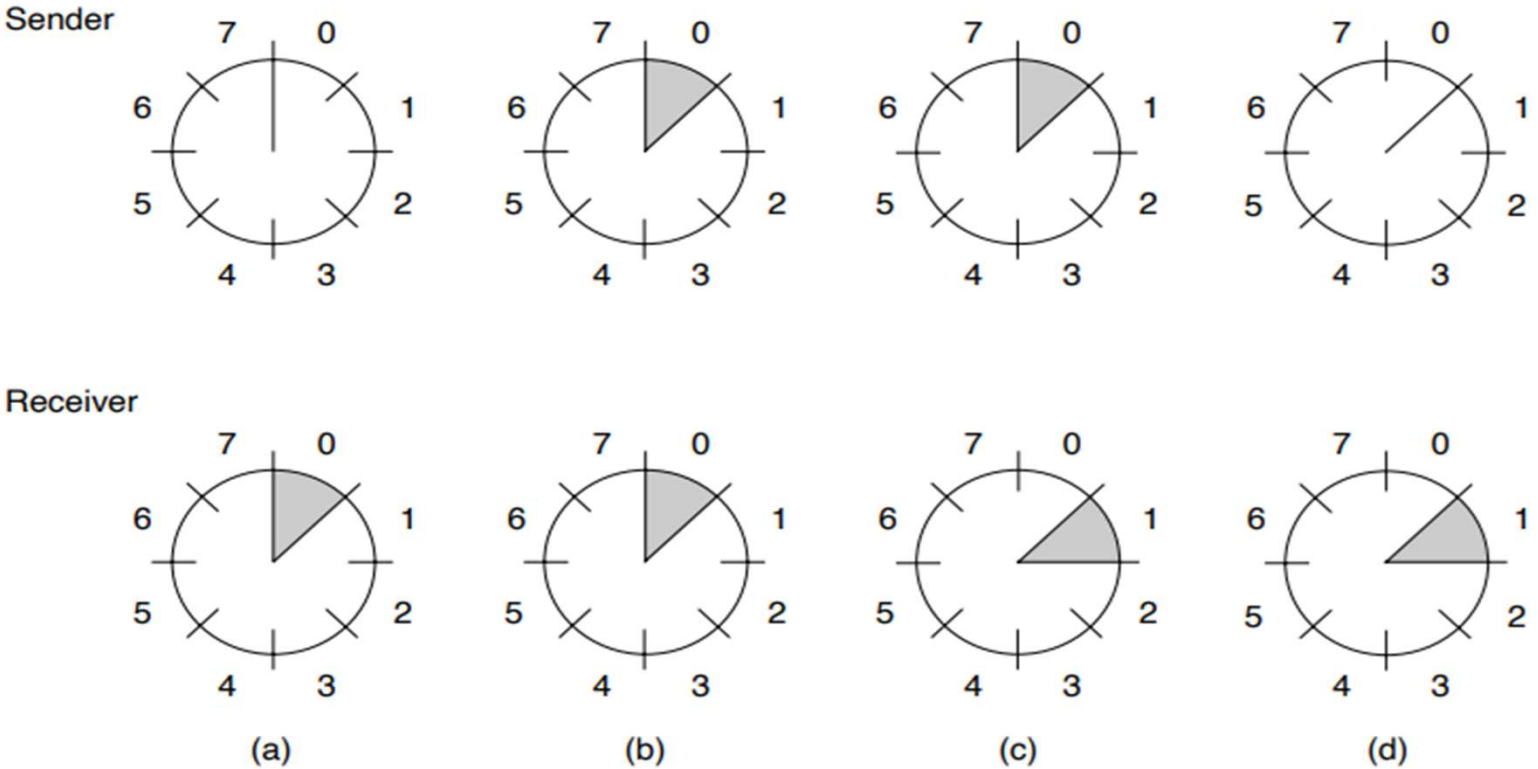

Figure 3-15. A sliding window of size 1, with a 3-bit sequence number. (a) Initially. (b) After the first frame has been sent. (c) After the first frame has been received. (d) After the first acknowledgement has been received.

3.4 IMPROVING EFFICIENCY

3.4.1 Goal: Bidirectional Transmission, Multiple Frames in Flight
Sliding Windows

- Because frames in the sender's window might be **lost or damaged**, the sender must **store all unacknowledged frames** in memory for possible retransmission.
- Therefore, if the **maximum window size is *n***, the sender must maintain ***n* frame buffers**.
- If the sender's window becomes **full**, the data link layer must **temporarily block the network layer** from providing new packets until space becomes available.
- The **receiver's window** tracks which sequence numbers it is **willing to accept**:
    - Frames **within** the window are buffered.
    - When the frame whose sequence number equals the **lower edge** of the window arrives, it is delivered to the network layer and the window **slides forward by one**.
- Frames **outside** the receiver's window are **discarded**.
- In every case (accepted or discarded), the receiver **sends an acknowledgement** so the sender knows how to proceed.
- A **window size of 1** forces strict in-order acceptance at the data link layer (stop-and-wait).
  Larger windows allow **out-of-order reception**, although…
- The **network layer always receives frames in correct order**, regardless of data link window size.
- Figure 3-15 (referenced) shows an example where:
    - Sender's window size is 1.
    - Initially, sender has **no outstanding frames** (window edges equal).
    - Over time, frames are sent, acknowledged, and the sender window moves.
    - The receiver's window **remains constant in size**, shifting as each expected frame arrives.

3.4 IMPROVING EFFICIENCY

3.4.2 Examples of Full-Duplex, Sliding Window Protocols

- A one-bit sliding window protocol is essentially **stop-and-wait**, since the sender can have only **one outstanding frame** at a time.
- It supports **retransmission of erroneous frames**, even when bidirectional communication is occurring.
- The protocol keeps two key variables:
  - **next_frame_to_send** – sequence number of the frame the sender is about to transmit (0 or 1).
  - **frame_expected** – sequence number the receiver is expecting next (also 0 or 1).
- Only **one** of the two communicating data link layers begins by sending the first frame.
  That machine executes **to_physical_layer()** and **start_timer()** before the main loop.
- The starting machine:
  - Fetches the first packet from the network layer.
  - Builds a frame containing this packet.
  - Sends it over the physical layer.
- When a frame arrives at the receiver:
  - The receiver checks whether it is a **duplicate** (same sequence number already accepted earlier).
  - If it matches **frame_expected**, the frame is:
    - Delivered to the network layer.
    - The receiver's window slides forward (frame_expected flips between 0 ↔ 1).

3.4 IMPROVING EFFICIENCY

3.4.2 Examples of Full-Duplex, Sliding Window Protocols

- Each frame sent includes:
  - A **sequence number** (seq).
  - An **acknowledgement field** (ack), indicating the last frame received correctly.
- When an acknowledgement arrives at the sender:
  - If the ack matches **next_frame_to_send**, the sender knows the previous frame was accepted and may fetch a **new packet** from the network layer.
  - Otherwise, the sender continues **retransmitting the same frame**.
- For every frame received, the receiver immediately **sends a frame back** (either data or pure acknowledgement).
- Pathological case considered:
  - Suppose A sends frame 0 to B, but A’s timeout is set too short.
  - A times out repeatedly before receiving B’s acknowledgement.
  - A retransmits **multiple copies** of frame 0 (seq = 0, ack = 1).
  - B must handle these duplicates correctly by ignoring repeated sequence numbers.

3.4.2 Examples of Full Duplex, Sliding Window Protocols
One Bit Sliding Window

```
/* Protocol 4 (Sliding window) is bidirectional. */

#define MAX_SEQ 1                                   /* must be 1 for protocol 4 */
typedef enum {frame_arrival, cksum_err, timeout} event_type;
#include "protocol.h"
void protocol4 (void)
{
  seq_nr next_frame_to_send;                        /* 0 or 1 only */
  seq_nr frame_expected;                            /* 0 or 1 only */
  frame r, s;                                       /* scratch variables */
  packet buffer;                                    /* current packet being sent */
  event_type event;

  next_frame_to_send = 0;                           /* next frame on the outbound stream */
  frame_expected = 0;                               /* frame expected next */
  from_network_layer(&buffer);                      /* fetch a packet from the network layer */
  s.info = buffer;                                  /* prepare to send the initial frame */
  s.seq = next_frame_to_send;                       /* insert sequence number into frame */
  s.ack = 1 – frame_expected;                       /* piggybacked ack */
  to_physical_layer(&s);                            /* transmit the frame */
  start_timer(s.seq);                               /* start the timer running */
```

```
while (true) {
    wait_for_event(&event);                    /* frame_arrival, cksum_err, or timeout */
    if (event == frame_arrival) {              /* a frame has arrived undamaged */
        from_physical_layer(&r);               /* go get it */
        if (r.seq == frame_expected) {         /* handle inbound frame stream */
            to_network_layer(&r.info);         /* pass packet to network layer */
            inc(frame_expected);               /* invert seq number expected next */
        }
        if (r.ack == next_frame_to_send) {     /* handle outbound frame stream */
            stop_timer(r.ack);                 /* turn the timer off */
            from_network_layer(&buffer);       /* fetch new pkt from network layer */
            inc(next_frame_to_send);           /* invert sender's sequence number */
        }
    }
    s.info = buffer;                           /* construct outbound frame */
    s.seq = next_frame_to_send;                /* insert sequence number into it */
    s.ack = 1 - frame_expected;                /* seq number of last received frame */
    to_physical_layer(&s);                     /* transmit a frame */
    start_timer(s.seq);                        /* start the timer running */
  }
}
```

Figure 3-16. A 1-bit sliding window protocol.

3.4 IMPROVING EFFICIENCY

3.4.2 Examples of Full-Duplex, Sliding Window Protocols

One-Bit Sliding Window

When computer B receives the **first valid frame**, it accepts it and updates **frame_expected = 1**.

All later incoming frames with **sequence number 0** are **rejected** because B is now expecting **sequence number 1**.

These rejected frames also contain **ack = 1**, but B is still waiting for **ack = 0**, so it **cannot fetch a new packet** from its network layer.

For each rejected duplicate frame, B sends back a frame with
seq = 0, ack = 0.

Eventually, one of these acknowledgement frames reaches A successfully, allowing A to **proceed to the next packet**.

Regardless of lost frames or premature timeouts, the protocol:

- Never delivers duplicate packets to the network layer.
- Never skips packets.
- Never deadlocks.
  → **The protocol is correct.**

A subtle issue appears if **both sides start by sending their first frame simultaneously**.
Fig. 3-17(a): If B waits for A's first frame before sending, communication behaves normally and **no duplicates occur**.

Fig. 3-17(b): If A and B transmit their first frames at the same time, the frames **cross in transit**, creating a synchronization anomaly:

- Many frames then appear as **duplicates** despite no transmission errors.
- Only half of the arriving frames carry new packets for the network layer.

Similar duplicate-frame situations can also occur with **premature timeouts**, even when one side clearly starts first.
Multiple premature timeouts may cause **three or more transmissions of the same frame**, wasting channel bandwidth.

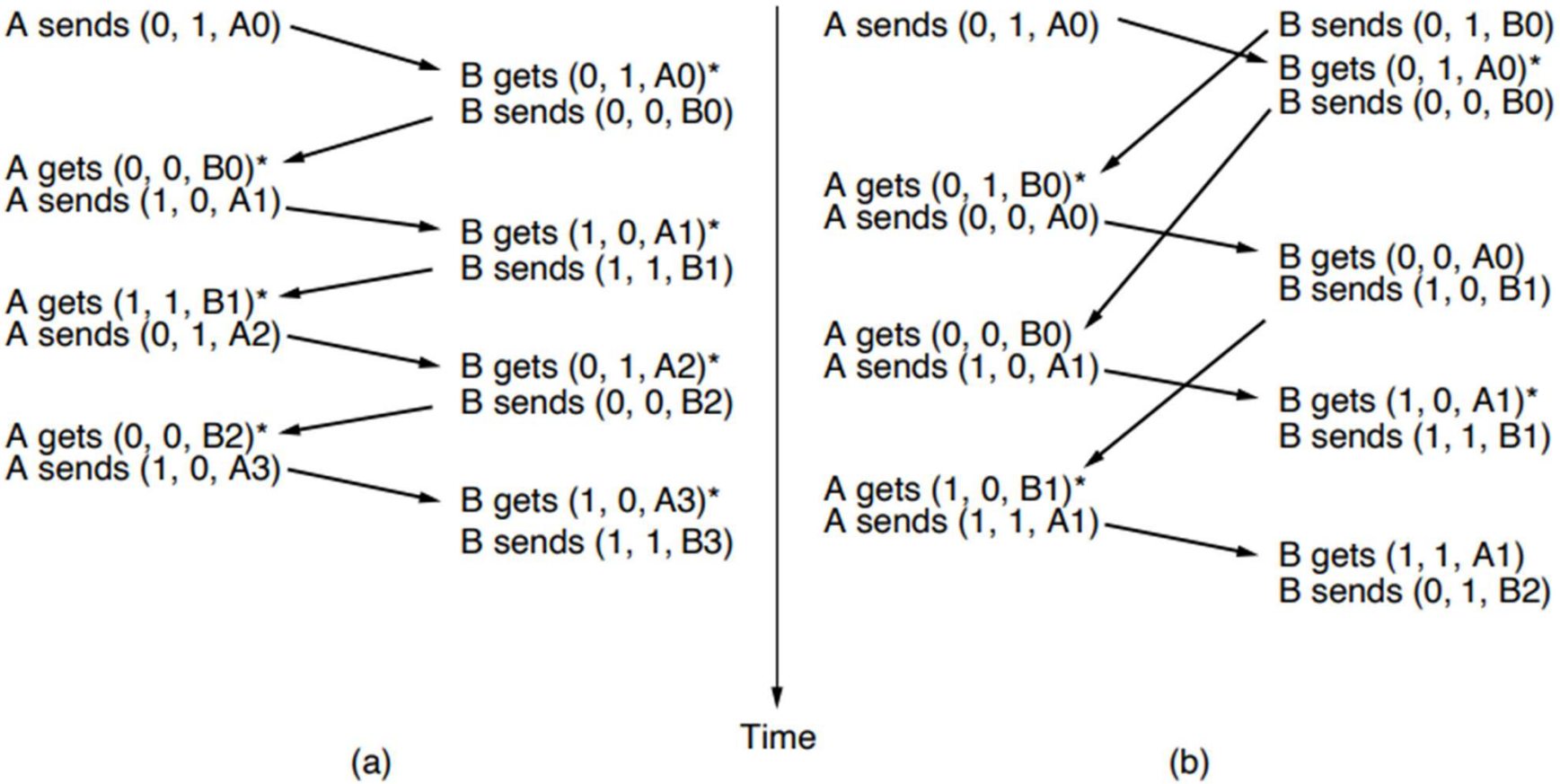

Figure 3-17. Two scenarios for protocol 4. (a) Normal case. (b) Abnormal case. The notation is (seq, ack, packet number). An asterisk indicates where a network layer accepts a packet.

3.4 IMPROVING EFFICIENCY

3.4.2 Examples of Full-Duplex, Sliding Window Protocols

Go-Back-N

Previous protocols assumed the round-trip time (RTT) was negligible.
→ This assumption breaks down for long-delay links (e.g., satellite).

Example scenario:

- Channel bandwidth: **50 kbps**
- One-way propagation delay: **250 ms**
- Round-trip time: **500 ms**
- Frame size: **1000 bits**

Timeline for sending **one** 1000-bit frame using stop-and-wait (Protocol 4):

- **t = 0 ms:** Sender begins transmitting the frame.
- **t = 20 ms:** Frame fully transmitted.
- **t = 270 ms:** Frame has arrived completely at the receiver.
- **t = 520 ms:** Acknowledgement arrives back at the sender.

Consequence:

- Sender is blocked for **500 of 520 ms → 96% idle time**.
- Only **4% of bandwidth is utilized**.
- Long delay + high bandwidth + small frame size = **very low efficiency**.

Root cause:

- Protocol forces the sender to **wait for each acknowledgement** before sending the next frame.

Solution:

- Allow the sender to send up to **w frames** before waiting (i.e., a sliding window).

If **w is large enough**, the sender transmits continuously:

- By the time the window becomes full, acknowledgements for earlier frames will have arrived.
- → Sender does **not block**; link utilization stays high.

To determine the required window size:

- Compute the **bandwidth-delay product (BD)**:
  - BD = (bandwidth in bits/sec) × (one-way propagation delay)
  - Represents how many bits are *in flight* at once.
- Convert BD to **number of frames** by dividing by bits per frame.

3.4 IMPROVING EFFICIENCY

Recommended window size:

- **w = 2BD + 1**
  - 2BD accounts for frames in flight during the RTT.
  - **+1** accounts for the fact that an acknowledgement is only sent after a full frame is received.
- For the example link:
- Bandwidth: **50 kbps**
- One-way delay: **250 ms**
- Frame size: **1000 bits**
- **Bandwidth-delay product (BD):**
  - BD = 50,000 bits/s × 0.25 s = **12,500 bits = 12.5 frames**
- **Window size requirement:**
  - w = **2BD + 1 = 26 frames**

Timeline with optimal window size (w = 26):

- Sender sends a new frame every **20 ms**.
- After sending 26 frames, at **t = 520 ms**, acknowledgement for frame 0 arrives.
- From that point:
  - Acknowledgements arrive every **20 ms**.
  - The sender gets permission to send a new frame exactly when needed.
  - **25–26 frames remain in flight continuously.**

$$\text{link utilization} \leq \frac{w}{1 + 2BD}$$

Result:

- With window size 26, link utilization becomes **100%**.
- Smaller window sizes lead to lower utilization because the sender becomes blocked.

Utilization formula:

- Utilization = fraction of time the sender is *not* blocked
- (Exact formula omitted in text, but concept emphasized.)

Key insight:

- Large **bandwidth-delay products** require **large window sizes** for efficient transmission.
  - High delay → sender has to wait long for ACKs.
  - High bandwidth → sender can transmit many frames before delay catches up.
  - In either case, a small window becomes a bottleneck.

Stop-and-wait comparison:

- w = 1 (stop-and-wait)
- If there is even **one frame's worth of propagation delay**, efficiency drops below **50%**.

Go Back N

For smaller window sizes, the utilization of the link will be less than 100% since the sender will be blocked sometimes. We can write the utilization as the fraction of time that the sender is not blocked:

$$\text{link utilization} \leq \frac{w}{1 + 2BD}$$

3.4 IMPROVING EFFICIENCY

3.4.2 Examples of Full-Duplex, Sliding Window Protocols

Go-Back-N

- **Pipelining frames** increases efficiency but introduces problems when frames are lost or damaged.
- **Main issue:**
  1. If a frame in the middle of the pipeline is lost or corrupted, many later frames may arrive at the receiver **before** the sender realizes something went wrong.
- **Receiver's obligation:**
  1. Must deliver packets to the network layer **in order**.
  2. Therefore, it cannot pass later frames (e.g., frame 3) until earlier ones (e.g., frame 2) are correctly received.

- **Two approaches** exist for dealing with errors in pipelined systems (shown in Fig. 3-18):
  1. **Go-Back-N**
  2. **Selective Repeat** (not summarized here, as text only discusses Go-Back-N)

Go-Back-N Strategy

- Receiver uses a **receive window of size 1**:
  - Accepts **only** the next expected frame.
  - **All subsequent frames are discarded**, even if they are correct.
  - Discarded frames receive **no acknowledgements (ACKs)**.
- **Sender behavior:**
  - Continues transmitting frames normally.
  - If the sender's window becomes full and no ACK arrives for the missing frame:
    - The sender eventually **times out**.
    - It then **retransmits all unacknowledged frames**, starting from the missing/damaged one.
- **Bandwidth impact:**
  - Wasteful when error rate is high because many correctly received frames are discarded and must be retransmitted.

3.4 IMPROVING EFFICIENCY

Example (Fig. 3-18a):

- Frames 0 and 1 are received successfully → ACKed.
- Frame 2 is lost or damaged.
- Sender continues sending frames 3, 4, … unaware of the problem.
- Receiver discards frames 3, 4, … because it is still waiting for frame 2.
- Sender eventually times out on frame 2.
- Sender then retransmits: **2, 3, 4, …** again.

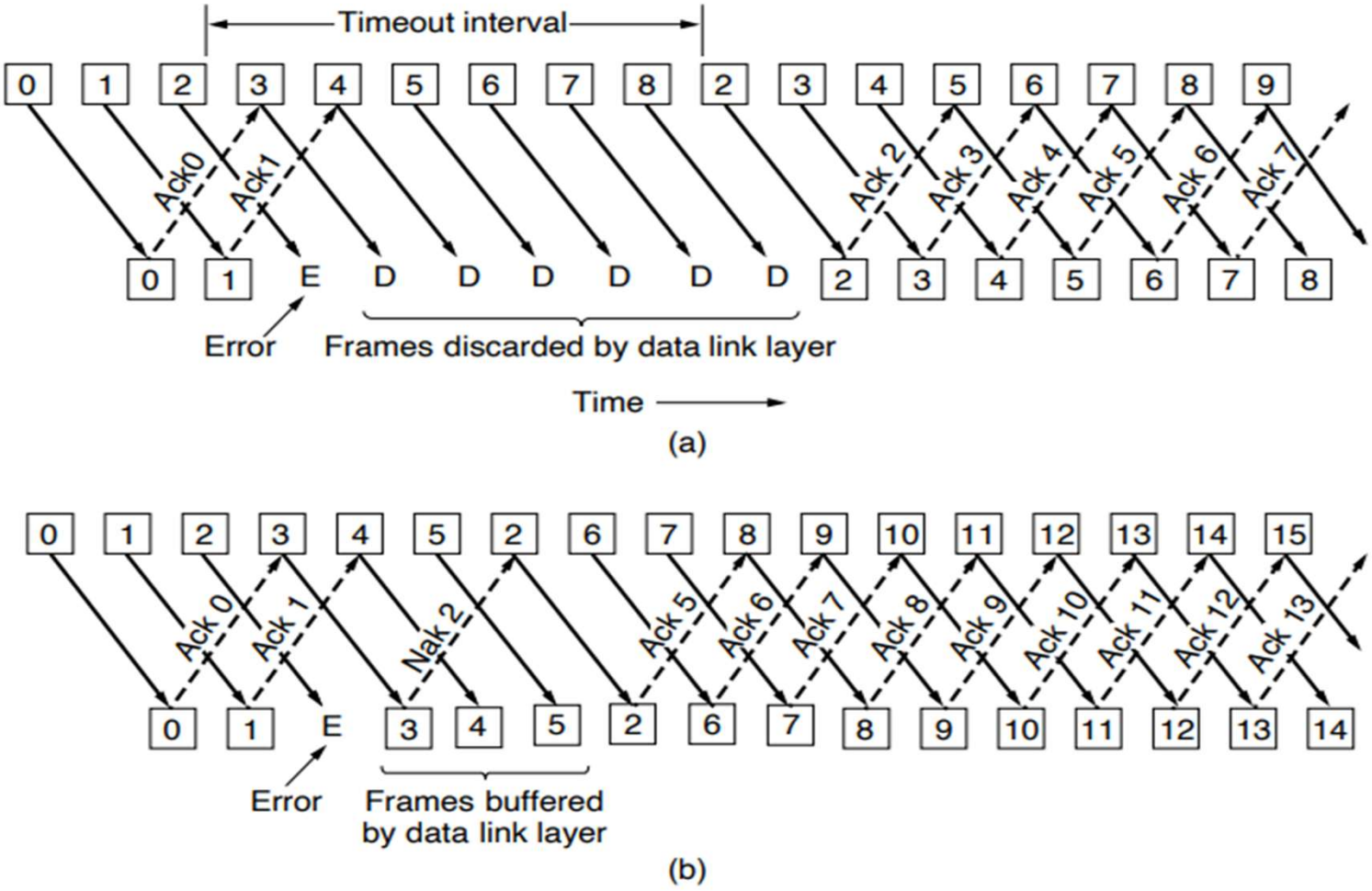

Figure 3-18. Pipelining and error recovery. Effect of an error when (a) receiver's window size is 1 and (b) receiver's window size is large.

3.4 IMPROVING EFFICIENCY

3.4.2 Examples of Full-Duplex, Sliding Window Protocols

Selective Repeat

Selective Repeat improves efficiency compared to go-back-n, especially on noisy channels with frequent errors.

Core idea:

- Receiver accepts and **buffers correctly received frames**, even if earlier frames were lost or damaged.
- Only the **faulty or missing frame** must be retransmitted.

Behavior on errors:

- A damaged or missing frame is discarded.
- Subsequent correct frames are **stored in a buffer**, not discarded.

Sender behavior:

- When its timer expires, the sender **retransmits only the oldest unacknowledged frame**, not the entire window.
- Once this missing frame is received, the receiver can deliver all buffered frames in proper order to the network layer.

Receiver window:

- Must be **larger than 1**, unlike go-back-n.
- Larger window ⇒ more buffering required.

Memory usage:

- Selective repeat may require **substantial data link layer memory** if the window is large.

3.4 IMPROVING EFFICIENCY

3.4.2 Examples of Full-Duplex, Sliding Window Protocols
Selective Repeat

Use of Negative Acknowledgements (NAKs)

- Often combined with NAKs for improved performance.
- Receiver sends a NAK when:
  - A checksum error occurs, or
  - A frame arrives **out of sequence**.
- NAKs prompt **early retransmission** before the timer expires.

Example (Fig. 3-18b):

- Frames 0 and 1 arrive correctly → ACKed.
- Frame 2 is lost.
- Receiver:
  - Receives frame 3 → detects missing frame → sends **NAK 2**, buffers frame 3.
  - Receives frames 4 and 5 → buffers them as well.
- Sender:
  - Receives NAK 2 → immediately retransmits frame 2.
- Once frame 2 arrives:
  - Receiver now has frames 2, 3, 4, 5.
  - All are **delivered in order** to the network layer.
  - Receiver acknowledges up through frame 5.
- If NAK is lost:
  - Sender will eventually **timeout for frame 2** and retransmit it anyway, but after a delay.

Trade-offs

- **Go-Back-N:**
  - Saves buffer space
  - Wastes bandwidth when errors are frequent
- **Selective Repeat:**
  - Saves bandwidth
  - Requires more buffer space

3.4 IMPROVING EFFICIENCY

3.4.2 Examples of Full-Duplex, Sliding Window Protocols
Selective Repeat

- Figure 3-19 shows a **go-back-n** protocol where the receiver **only accepts frames in order**; any frames after an error are discarded.
- For the first time in these protocols, the **network layer may not always have packets available**.
    - When it does have a packet, it triggers a **network layer ready** event.
    - To enforce flow control, the data link layer can **enable** or **disable** the network layer from sending more packets.

Outstanding Frames vs. Sequence Number Space

- Important distinction:
  Maximum outstanding frames ≠ sequence number space size.
- In go-back-n:
    - Up to **MAX_SEQ** frames may be outstanding simultaneously.
    - But there are **MAX_SEQ + 1** distinct sequence numbers:
      `0, 1, 2, …, MAX_SEQ`.
- This difference becomes **more restrictive** in selective repeat.

3.4 IMPROVING EFFICIENCY

3.4.2 Examples of Full-Duplex, Sliding Window Protocols Selective Repeat

Why the Restriction Is Needed (Example with MAX_SEQ = 7)

Scenario:

1. Sender transmits frames **0–7**.
2. Receiver sends a piggybacked **ACK for 7**, which arrives at sender.
3. Sender transmits **another batch** of eight frames, again numbered **0–7** (sequence numbers wrap around).
4. Another piggybacked **ACK for frame 7** arrives.

- **Ambiguity:**
  The sender cannot tell whether:
  - All eight frames of the *second* batch arrived correctly, **or**
  - All eight frames were **lost**, so the receiver is still acknowledging the *first* batch.
- Because the **same ACK number (7)** is valid in both cases, the sender has **no way to distinguish** them.

Conclusion of the Example

- To avoid ambiguity, the sender must **not allow MAX_SEQ + 1 frames** to be outstanding.
- The **maximum number of outstanding frames must be ≤ MAX_SEQ.**
- This rule prevents wrapping sequence numbers from causing confusion between different batches of frames.

Sender Buffering Requirements

- Even though **protocol 5** (the version shown) does not buffer frames received after an error:
  - The **sender must still buffer all unacknowledged frames**.
  - Reason: If a timeout occurs, it may need to **retransmit all outstanding frames**.

Selective Repeat

```
/* Protocol 5 (Go-back-n) allows multiple outstanding frames. The sender may transmit up
   to MAX_SEQ frames without waiting for an ack. In addition, unlike in the previous
   protocols, the network layer is not assumed to have a new packet all the time. Instead,
   the network layer causes a network_layer_ready event when there is a packet to send. */

#define MAX_SEQ 7

typedef enum {frame_arrival, cksum_err, timeout, network_layer_ready} event_type;

#include "protocol.h"


static boolean between(seq_nr a, seq_nr b, seq_nr c)
{
/* Return true if a <= b < c circularly; false otherwise. */
  if (((a <= b) && (b < c)) || ((c < a) && (a <= b)) || ((b < c) && (c < a)))
       return(true);
    else
       return(false);
}
```

```
static void send_data(seq_nr frame_nr, seq_nr frame_expected, packet buffer[])
{
/* Construct and send a data frame. */
  frame s;                                        /* scratch variable */
  s.info = buffer[frame_nr];                      /* insert packet into frame */
  s.seq = frame_nr;                               /* insert sequence number into frame */
  s.ack = (frame_expected + MAX_SEQ) % (MAX_SEQ + 1); /* piggyback ack */
  to_physical_layer(&s);                          /* transmit the frame */
  start_timer(frame_nr);                          /* start the timer running */
}

void protocol5(void)
{
  seq_nr next_frame_to_send;                      /* MAX_SEQ > 1; used for outbound stream */
  seq_nr ack_expected;                            /* oldest frame as yet unacknowledged */
  seq_nr frame_expected;                          /* next frame expected on inbound stream */
  frame r;                                        /* scratch variable */
  packet buffer[MAX_SEQ + 1];                     /* buffers for the outbound stream */
  seq_nr nbuffered;                               /* number of output buffers currently in use */
  seq_nr i;                                       /* used to index into the buffer array */
  event_type event;
  enable_network_layer();                         /* allow network_layer_ready events */
  ack_expected = 0;                               /* next ack expected inbound */
  next_frame_to_send = 0;                         /* next frame going out */
  frame_expected = 0;                             /* number of frame expected inbound */
  nbuffered = 0;                                  /* initially no packets are buffered */
  while (true) {
    wait_for_event(&event);                       /* four possibilities: see event_type above */
```

```
    switch(event) {
      case network_layer_ready:                 /* the network layer has a packet to send */
           /* Accept, save, and transmit a new frame. */
           from_network_layer(&buffer[next_frame_to_send]); /* fetch new packet */
           nbuffered = nbuffered + 1;           /* expand the sender's window */
           send_data(next_frame_to_send, frame_expected, buffer);/* transmit the frame */
           inc(next_frame_to_send);             /* advance sender's upper window edge */
           break;
      case frame_arrival:                       /* a data or control frame has arrived */
           from_physical_layer(&r);             /* get incoming frame from physical layer */
           if (r.seq == frame_expected) {
                /* Frames are accepted only in order. */
                to_network_layer(&r.info);      /* pass packet to network layer */
                inc(frame_expected);            /* advance lower edge of receiver's window */
            }
            /* Ack n implies n - 1, n - 2, etc.  Check for this. */
           while (between(ack_expected, r.ack, next_frame_to_send)) {
                /* Handle piggybacked ack. */
                nbuffered = nbuffered - 1;      /* one frame fewer buffered */
                stop_timer(ack_expected);       /* frame arrived intact; stop timer */
                inc(ack_expected);              /* contract sender's window */
           }
           break;
      case cksum_err: break;                    /* just ignore bad frames */
      case timeout:                             /* trouble; retransmit all outstanding frames */
           next_frame_to_send = ack_expected;       /* start retransmitting here */
           for (i = 1; i <= nbuffered; i++) {
                send_data(next_frame_to_send, frame_expected, buffer);/* resend frame */
                inc(next_frame_to_send);        /* prepare to send the next one */
           }
    }
    if (nbuffered < MAX_SEQ)
           enable_network_layer();
    else
           disable_network_layer();
  }
}
```

Figure 3-19. A sliding window protocol using go-back-n.

3.4 IMPROVING EFFICIENCY

3.4.2 Examples of Full-Duplex, Sliding Window Protocols

Selective Repeat

- **Cumulative Acknowledgements**
  - When ACK for frame **n** arrives, it implicitly acknowledges **all frames before it** (n−1, n−2, …).
  - Helps recover from lost or corrupted ACK frames.
  - On receiving an ACK, the sender checks which buffer slots can now be freed.
- **Window Management**
  - When buffers are freed, and space becomes available in the sender window, a previously blocked network layer may be **re-enabled** to send more packets.
- **Piggybacking Assumption**
  - Protocol 5 assumes **continuous reverse traffic**, allowing ACKs to be piggybacked.
  - Protocol 4 does **not** require this because it sends a frame back for every frame received.

Timers in Selective Repeat

- Because multiple frames can be outstanding, **each transmitted frame needs its own timer**.
- Each timer operates **independently**, ensuring only the timed-out frame is retransmitted.
- Instead of multiple hardware timers, **one hardware clock** can simulate multiple timers.

3.4 IMPROVING EFFICIENCY

3.4.2 Examples of Full-Duplex, Sliding Window Protocols

Selective Repeat

Simulating Multiple Timers (Fig. 3-20 Example)

- The system uses:
  - A single hardware clock that ticks periodically (e.g., every 1 ms).
  - A **linked list of pending timeouts**, ordered by expiry time.
- Each node in the timeout list contains:
  - The number of clock ticks remaining before expiration.
  - The frame number being timed.
  - A pointer to the next node.
- Operation:
  - At each clock tick, the timeout counter of the **first** node is decremented.
  - When it reaches **zero**, a timeout event is triggered and the node is removed.
  - This approach minimizes per-tick work even though start/stop timer operations require scanning the list.
- In protocol 5, both `start_timer(frame)` and `stop_timer(frame)` specify **which frame** the timer corresponds to.

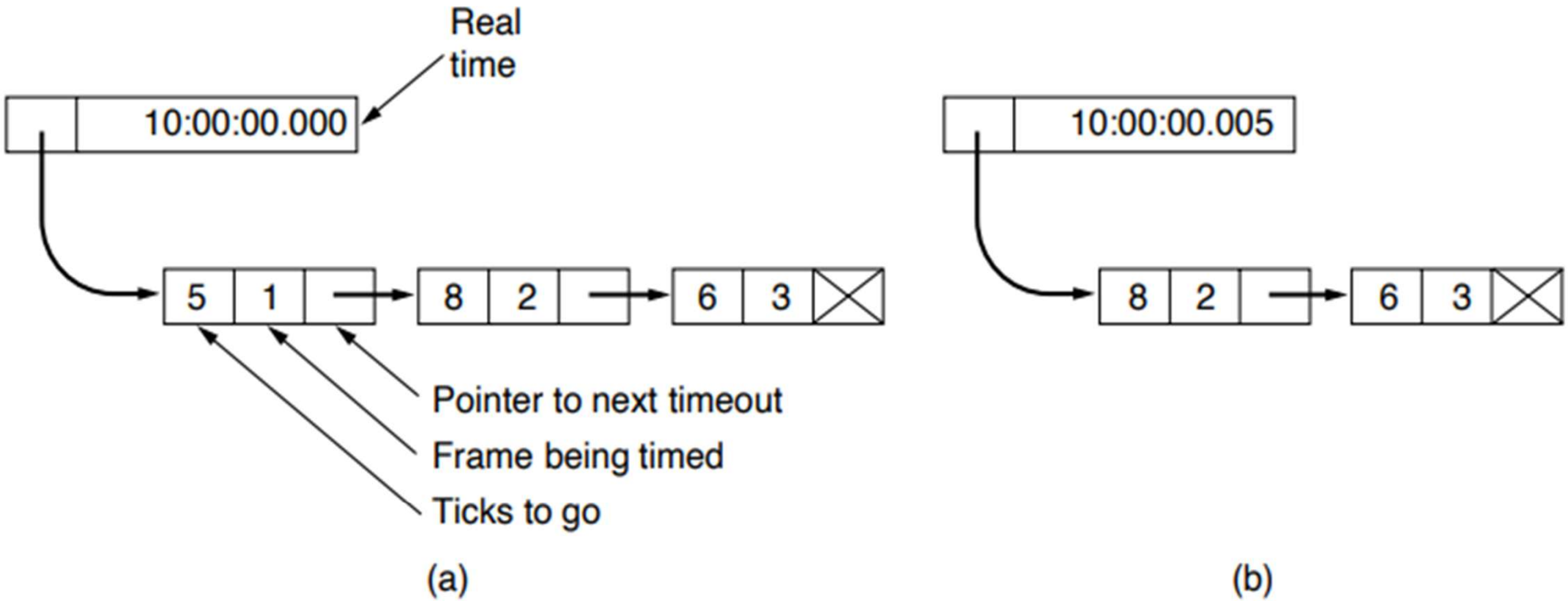

Figure 3-20. Simulation of multiple timers in software. (a) The queued timeouts. (b) The situation after the first timeout has expired.

3.4 IMPROVING EFFICIENCY

3.4.2 Examples of Full-Duplex, Sliding Window Protocols
Selective Repeat

- **Both sender and receiver maintain windows**:
  - **Sender window**:
    - Starts at size **0**, grows up to a **predefined maximum**.
    - Contains unacknowledged frames waiting for delivery.
  - **Receiver window**:
    - **Fixed size**, equal to the chosen maximum window.
    - Represents the range of sequence numbers the receiver is willing to accept.

Receiver Buffers & Acceptance Rules

- The receiver maintains:
  - **One buffer per sequence number** in its window.
  - An **arrived bit** for each buffer (full/empty status).
- When a frame arrives:
  - The receiver checks if its sequence number is **within the receiver window** (using `between()`).
  - If within the window **and** not previously received:
    - The frame is accepted and stored.
  - Acceptance is **independent of ordering**—the receiver may accept frames even if earlier frames haven't arrived.
- Frames are **not delivered to the network layer** until:
  - **All lower-numbered frames** have already been delivered.
  - Delivery is **in-sequence only**, even though reception is out of order.

Sequence Number Constraints

- Nonsequential reception imposes stricter constraints on sequence number space.
- Example:
  - With a **3-bit sequence number**, there are **8 numbers (0–7)**.
  - The sender may transmit up to **7 outstanding frames** before waiting for an ACK.

3.4 IMPROVING EFFICIENCY

3.4.2 Examples of Full-Duplex, Sliding Window Protocols Selective Repeat

Example Scenario (Fig. 3-22)

Initial State

- Sender and receiver windows both cover frames **0–6** (window size 7).
- Sender transmits frames **0, 1, 2, 3, 4, 5, 6** sequentially.

Receiver Behavior

- Receiver accepts all 7 frames correctly.
- It acknowledges them and **advances its window** so that it now expects:
  7, 0, 1, 2, 3, 4, 5
- All receiver buffers are marked **empty** again.

Disaster Scenario

- A lightning strike destroys **all acknowledgments** sent by the receiver.
- Sender eventually **times out** and retransmits **frame 0**.

Critical Problem

- When frame 0 arrives:
  - The receiver checks its window (7,0,1,2,3,4,5).
  - **Frame 0 is *inside* the new window**, so the receiver incorrectly treats it as a **new** frame.
  - It accepts the frame again and sends an ACK for **frame 6**, thinking it has frames 0–6 again.

```
/* Protocol 6 (Selective repeat) accepts frames out of order but passes packets to the
   network layer in order. Associated with each outstanding frame is a timer. When the timer
   expires, only that frame is retransmitted, not all the outstanding frames, as in protocol 5. */
#define MAX_SEQ 7                                        /* should be 2^n – 1 */
#define NR_BUFS ((MAX_SEQ + 1)/2)
typedef enum {frame_arrival, cksum_err, timeout, network_layer_ready, ack_timeout} event_type;
#include "protocol.h"
boolean no_nak = true;                                   /* no nak has been sent yet */
seq_nr oldest_frame = MAX_SEQ + 1;                       /* initial value is only for the simulator */

static boolean between(seq_nr a, seq_nr b, seq_nr c)
{
/* Same as between in protocol 5, but shorter and more obscure. */
  return ((a <= b) && (b < c)) || ((c < a) && (a <= b)) || ((b < c) && (c < a));
}

static void send_frame(frame_kind fk, seq_nr frame_nr, seq_nr frame_expected, packet buffer[])
{
/* Construct and send a data, ack, or nak frame. */
  frame s;                                               /* scratch variable */
  s.kind = fk;                                           /* kind == data, ack, or nak */
  if (fk == data) s.info = buffer[frame_nr % NR_BUFS];
  s.seq = frame_nr;                                      /* only meaningful for data frames */
  s.ack = (frame_expected + MAX_SEQ) % (MAX_SEQ + 1);
  if (fk == nak) no_nak = false;                         /* one nak per frame, please */
  to_physical_layer(&s);                                 /* transmit the frame */
  if (fk == data) start_timer(frame_nr % NR_BUFS);
  stop_ack_timer();                                      /* no need for separate ack frame */
}
```

```
void protocol6(void)
{
  seq_nr ack_expected;                          /* lower edge of sender's window */
  seq_nr next_frame_to_send;                    /* upper edge of sender's window + 1 */
  seq_nr frame_expected;                        /* lower edge of receiver's window */
  seq_nr too_far;                               /* upper edge of receiver's window + 1 */
  int i;                                        /* index into buffer pool */
  frame r;                                      /* scratch variable */
  packet out_buf[NR_BUFS];                      /* buffers for the outbound stream */
  packet in_buf[NR_BUFS];                       /* buffers for the inbound stream */
  boolean arrived[NR_BUFS];                     /* inbound bit map */
  seq_nr nbuffered;                             /* how many output buffers currently used */
  event_type event;
  enable_network_layer();                       /* initialize */
  ack_expected = 0;                             /* next ack expected on the inbound stream */
  next_frame_to_send = 0;                       /* number of next outgoing frame */
  frame_expected = 0;
  too_far = NR_BUFS;
  nbuffered = 0;                                /* initially no packets are buffered */
  for (i = 0; i < NR_BUFS; i++) arrived[i] = false;
  while (true) {
    wait_for_event(&event);                     /* five possibilities: see event_type above */
```

```
switch(event) {
  case network_layer_ready:                    /* accept, save, and transmit a new frame */
       nbuffered = nbuffered + 1;              /* expand the window */
       from_network_layer(&out_buf[next_frame_to_send % NR_BUFS]); /* fetch new packet */
       send_frame(data, next_frame_to_send, frame_expected, out_buf);/* transmit the frame */
       inc(next_frame_to_send);                /* advance upper window edge */
       break;

  case frame_arrival:                          /* a data or control frame has arrived */
       from_physical_layer(&r);                /* fetch incoming frame from physical layer */
       if (r.kind == data) {
            /* An undamaged frame has arrived. */
            if ((r.seq != frame_expected) && no_nak)
               send_frame(nak, 0, frame_expected, out_buf); else start_ack_timer();
            if (between(frame_expected,r.seq,too_far) && (arrived[r.seq%NR_BUFS]==false)) {
                 /* Frames may be accepted in any order. */
                 arrived[r.seq % NR_BUFS] = true;/* mark buffer as full */
                 in_buf[r.seq % NR_BUFS] = r.info;/* insert data into buffer */
                 while (arrived[frame_expected % NR_BUFS]) {
                      /* Pass frames and advance window. */
                      to_network_layer(&in_buf[frame_expected % NR_BUFS]);
                      no_nak = true;
                      arrived[frame_expected % NR_BUFS] = false;
                      inc(frame_expected);     /* advance lower edge of receiver's window */
                      inc(too_far);            /* advance upper edge of receiver's window */
                      start_ack_timer();       /* to see if a separate ack is needed */
                 }
            }
       }
```

```
            if((r.kind==nak) && between(ack_expected,(r.ack+1)%(MAX_SEQ+1),next_frame_to_s
                send_frame(data, (r.ack+1) % (MAX_SEQ + 1), frame_expected, out_buf);
            while (between(ack_expected, r.ack, next_frame_to_send)) {
                nbuffered = nbuffered – 1;              /* handle piggybacked ack */
                stop_timer(ack_expected % NR_BUFS);     /* frame arrived intact */
                inc(ack_expected);                      /* advance lower edge of sender's window */
            }
            break;

        case cksum_err:
            if (no_nak) send_frame(nak, 0, frame_expected, out_buf); /* damaged frame */
            break;

        case timeout:
            send_frame(data, oldest_frame, frame_expected, out_buf); /* we timed out */
            break;

        case ack_timeout:
            send_frame(ack,0,frame_expected, out_buf);      /* ack timer expired; send ack */
    }

    if (nbuffered < NR_BUFS) enable_network_layer(); else disable_network_layer();
  }
}
```

Figure 3-21. A sliding window protocol using selective repeat.

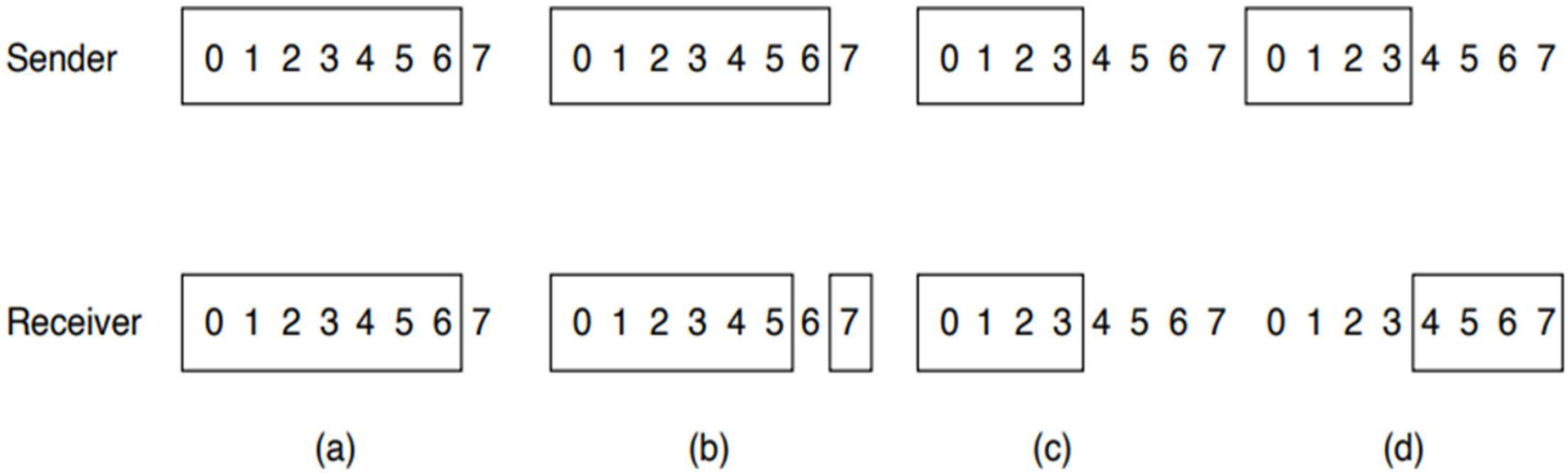

Figure 3-22. (a) Initial situation with a window of size7. (b) After 7 frames have been sent and received but not acknowledged. (c) Initial situation with a window size of 4. (d) After 4 frames have been sent and received but not acknowledged.

3.4 IMPROVING EFFICIENCY

3.4.2 Examples of Full-Duplex, Sliding Window Protocols
Selective Repeat

Sender Reaction & Failure Scenario

- The sender receives the (delayed) ACK for frame 6 and **believes all earlier frames were successfully delivered**.
- The sender **advances its window** and sends the next sequence of frames: **7, 0, 1, 2, 3, 4, 5**.
- The receiver:
    - Accepts **frame 7**, delivers it to the network layer.
    - Then checks for the next in-order frame and finds an **old, previously buffered frame 0**.
    - Mistakenly **delivers the old frame 0 again** as if it were new.
- Result:
  Network layer receives incorrect duplicate data → protocol failure.

Root Cause: Window Overlap

- After advancing the receiver window, the **new valid window overlaps the old window**.
- Therefore, arriving frames in the overlapping region (e.g., 0–5) could be:
    - **New frames** (if ACKs were received), or
    - **Old retransmissions** (if ACKs were lost).
- The receiver has **no way to distinguish duplicates from new frames** in overlapping regions.

Solution: Limit Window Size

- To avoid ambiguity, the receiver window must be advanced so that **new window does not overlap the old window**.
- This requires the **maximum window size ≤ half the sequence number space**.
- For a sequence space of size **MAX_SEQ + 1**, the window size must be:
  Window size = (MAX_SEQ + 1) / 2

3.4 IMPROVING EFFICIENCY

3.4.2 Examples of Full-Duplex, Sliding Window Protocols
Selective Repeat

Example With 3-Bit Sequence Numbers

- Sequence space: **0–7** (8 numbers).
- Maximum allowed window size: **4** (half of 8).
- With window size = 4:
    - If receiver accepts frames **0–3** and advances, the next valid window is **4–7**.
    - Frames 0–3 (old) and 4–7 (new) **never overlap**, avoiding ambiguity.

Receiver Buffer Requirement

- The receiver **never accepts frames outside its window**.
- Therefore, the number of buffers needed = **window size**, not sequence space size.
- In the 3-bit example:
    - Window size = **4** → receiver needs **4 buffers** (buffer 0, 1, 2, 3).
    - A frame numbered **i** is stored in buffer **i mod 4**.
    - i and (i + 4) map to the same buffer but **never occur in the window simultaneously**, since that would require a window size ≥ 5.

Timers

- The number of timers needed = number of buffers = window size.
- Each buffer has a corresponding timer:
    - When the timer expires, the frame stored in that buffer is **retransmitted**.

3.4 IMPROVING EFFICIENCY

3.4.2 Examples of Full-Duplex, Sliding Window Protocols

Selective Repeat

Relaxing the Heavy Reverse-Traffic Assumption

- Protocol 6 removes the assumption that there will always be frequent reverse-direction traffic for piggybacking acknowledgements.
- In Protocol 5, acknowledgements depended on reverse traffic; if none occurred, ACKs could be delayed too long.
- With heavy one-direction traffic and no reverse traffic, the sender's window could fill up, causing the protocol to **block**.

Auxiliary ACK Timer

- Protocol 6 introduces an **auxiliary acknowledgement timer**.
- When an in-sequence frame arrives, `start_ack_timer` is called.
- If no reverse traffic arrives before the timer expires, a **standalone ACK frame** is transmitted.
- This ensures ACKs are sent even when the reverse channel is idle.
- Only **one** auxiliary timer exists:
    - Calling `start_ack_timer` again while it is running has **no effect**.
    - The timer is not reset—its purpose is to maintain a **minimum ACK rate**.
- The auxiliary ACK timeout must be **shorter** than the sender's data-frame timeout to avoid unnecessary retransmissions.

3.4 IMPROVING EFFICIENCY

3.4.2 Examples of Full-Duplex, Sliding Window Protocols

Selective Repeat

Improved Error Handling with NAKs

- Protocol 6 uses a more efficient mechanism for error detection: **Negative Acknowledgements (NAKs)**.
- The receiver sends a NAK whenever it suspects an error.
- Two situations indicate suspicion:
  - **Damaged frame** arrives (checksum error).
  - **Out-of-order frame** arrives, indicating a potential lost frame.
- The NAK requests retransmission of a specific missing frame.

Avoiding Duplicate NAKs

- To prevent repeated requests for the same missing frame:
  - The receiver tracks whether a NAK has already been sent for the expected frame.
  - The variable **no_nak** is true when **no NAK** has yet been sent for the currently expected frame.
  - A NAK is only sent if `no_nak` is true.

3.4 IMPROVING EFFICIENCY

3.4.2 Examples of Full-Duplex, Sliding Window Protocols

Selective Repeat

Lost or Mangled NAKs

- If a NAK is lost or corrupted, no serious problem occurs.
  - The sender will eventually **timeout** and retransmit the missing frame anyway.
- If another **wrong frame** arrives *after* a NAK has been sent (and lost):
  - `no_nak` will be **true**.
  - The **auxiliary ACK timer** is started.
  - When it expires, an **ACK is sent**, which helps **resynchronize** sender and receiver.

Timer Tightness vs. Round-Trip Time Variability

- If propagation, processing, and ACK return delays are **nearly constant**:
  - Sender can use a **tight timer**, just slightly greater than the typical round-trip time.
  - **NAKs are not beneficial** in this scenario.
- If the round-trip time is **highly variable**:
  - e.g., reverse traffic is sporadic → ACK arrival time varies.
  - Sender must choose between:
    - **Small timeout** → many unnecessary retransmissions.
    - **Large timeout** → long idle periods after errors.
  - Both choices waste bandwidth.
- When **ACK timing variability (standard deviation)** is large:
  - Timer should be set **loose** (conservative).
  - **NAKs help** by speeding up recovery from lost or damaged frames.

3.4 IMPROVING EFFICIENCY

3.4.2 Examples of Full-Duplex, Sliding Window Protocols

Selective Repeat

Identifying Which Frame Timed Out (Protocol 6 vs Protocol 5)

- **Protocol 5**:
  - Always the oldest unacknowledged frame → `ack_expected`.
- **Protocol 6**:
  - Not obvious which frame timed out because:
    - Multiple frames can be outstanding.
    - They have **independent timers**.
- Example:
  - Outstanding frames: **0 1 2 3 4** (oldest → newest).
  - Events occur:
    - Frame **0** times out → retransmitted as 5 enters window.
    - Frame **1** times out.
    - Frame **2** times out → frame 6 transmitted.
  - Outstanding frames become: **3 4 0 5 1 2 6** (oldest → newest).
- If all ACKs are lost for a period:
  - Timers will expire in the order: 3, 4, 0, 5, 1, 2, 6.
- To manage this complexity:
  - Variable **`oldest_frame`** is set upon timeout to record **which frame** triggered the timeout event.