The Data Link Layer

Dr. Xiqun Lu

College of Computer Science

Zhejiang University

Outline

- Overview of Data link layer
- Data link layer: **Framing**
- Data link layer: **Error Control**
  - **Error Correction**
  - **Error Detection**
- Elementary data link protocols
- **Sliding window protocols**
- Examples of data link protocols

Outline

- Overview of Data link layer
- Data link layer: **Framing**
- Data link layer: **Error Control**
  - **Error Correction**
  - **Error Detection**
- Elementary data link protocols
- **Sliding window protocols**
- Examples of data link protocols

The Design Principles of the Data Link Layer

- The essential property of a channel that makes it “wire-like” is that the bits are delivered in exactly the same order in which they are sent.
  - To achieve **reliable** and **efficient** communication between two adjacent machines.
- The data link layer uses the services of the physical layer to send and receive bits over communication channels.
- The **characteristics** of general communication channels are:
  - Errors
  - Finite data rate
  - Propagation Delay
- Three main functions of the data link layer
  - Framing
  - Dealing with errors (error detection and error correction)
  - Flow control

Design issues: protocols

- Framing
  - Break stream of bits up into discrete frames
- Error control (Error Detection and Error Correction)
  - How does a sender know that all packets are correctly received
    - Acknowledgement, timer and sequence numbers
- Flow control
  - How to prevent a sender to overload the receiver with packets
    - Feedback-based flow control (the data link layer): receiver gives sender permission
    - Rate-based flow control (the transportation layer): limit the data rate at which the sender may transmit data.

Position of the Data Link Layer

- The function of the data link layer is to provide services to the network layer.

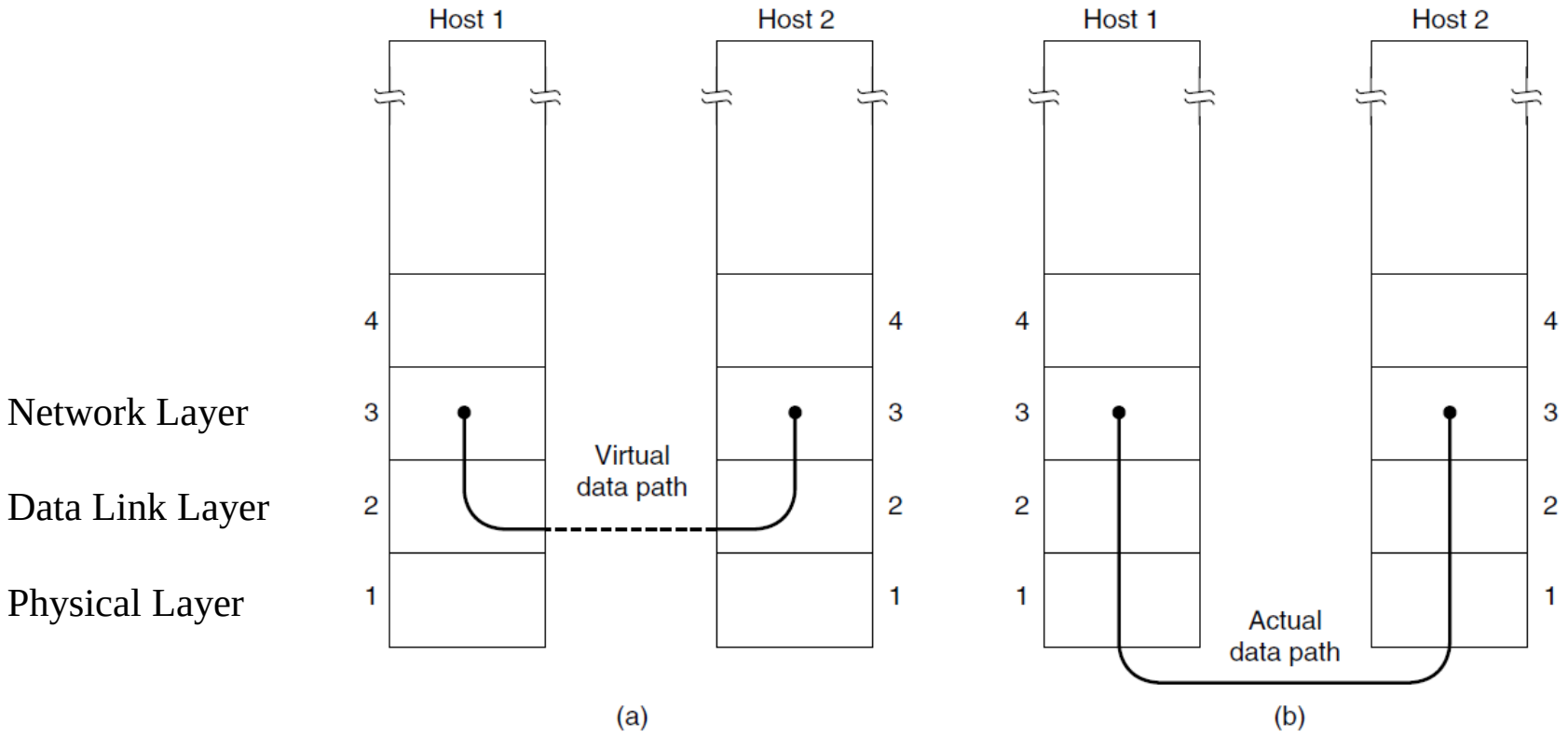
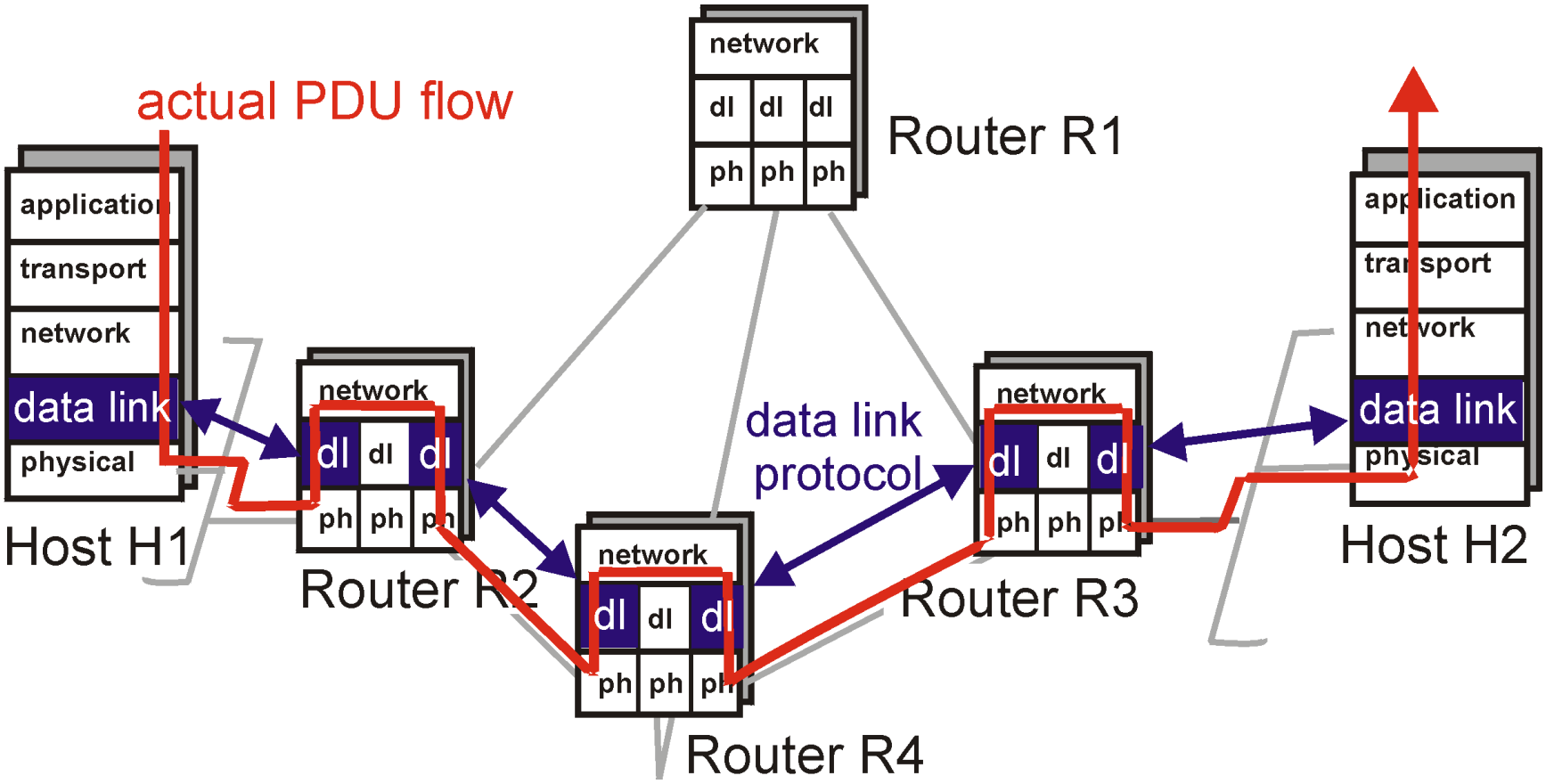


Figure 3-2. (a) Virtual communication. (b) Actual communication.

Position of the Data Link Layer



Where is the Link Layer Implemented? [4]

- In a router, the link layer is implemented in the router's line card.
- Is a host's link layer implemented in hardware or software?
  - Most of the link-layer is implemented in hardware on the network adapter (the interface). Part of the link layer is implemented in software that runs on the host's CPU.

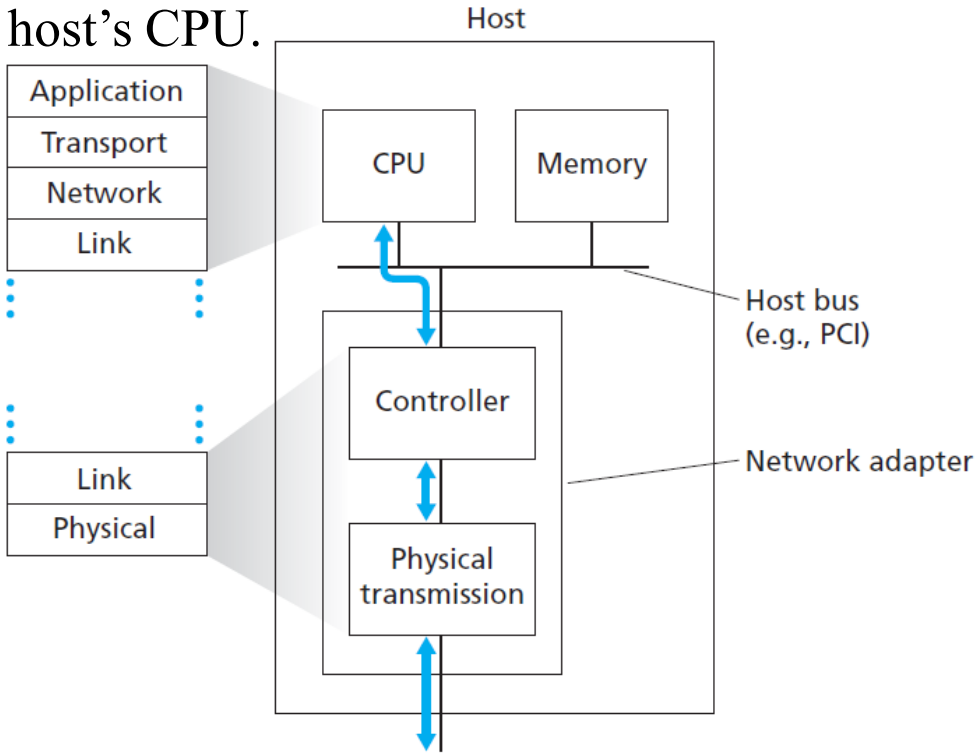


Figure 5.2 ♦ Network adapter: its relationship to other host components and to protocol stack functionality

For the most part, the link layer is implemented in a network adapter, also sometimes known as a network interface card (**NIC**). At the heart of the network adapter is the link-layer controller, usually a single, special-purpose chip that implements many of the link-layer services (frame, link access, flow control, error detection, etc.)

The Services of Data Link Layer (I)

- The actual services that are offered vary from protocol to protocol.
- 1) Unacknowledged connectionless service
  - To send independent frames to the destination without having acknowledge.
  - This class of service is appropriate for very low error-rate and real-time applications
  - Example: Ethernet
- 2) Acknowledged connectionless service
  - There is no logical connections, but each frame sent is individually acknowledged.
  - This service is useful for unreliable channels.
  - Example: 802.11 WiFi

The Services of Data Link Protocol (II)

- 3) Acknowledged connection-oriented service
  - The source and the destination machines establish a connection before any data are transferred. This service has three distinct phases
    - To establish the connection
    - To transmit data frames
    - To release the connection
  - This kind of service is appropriate over long, unreliable links.
  - Example: a satellite channel or a long-distance telephone circuit.

Outline

- Overview of Data link layer
- Data link layer: **Framing**
- Data link layer: **Error Control**
  - **Error Correction**
  - **Error Detection**
- Elementary data link protocols
- **Sliding window protocols**
- Examples of data link protocols

Outline

- Overview of Data link layer
- Data link layer: **Framing**
- Data link layer: **Error Control**
  - **Error Correction**
  - **Error Detection**
- Elementary data link protocols
- **Sliding window protocols**
- Examples of data link protocols

Data Link Layer: Framing

- The data link layer takes the packets it get from the network layer and encapsulates them into **frames** for transmission.

- General Frame Format: a header, a payload field, and a trailer

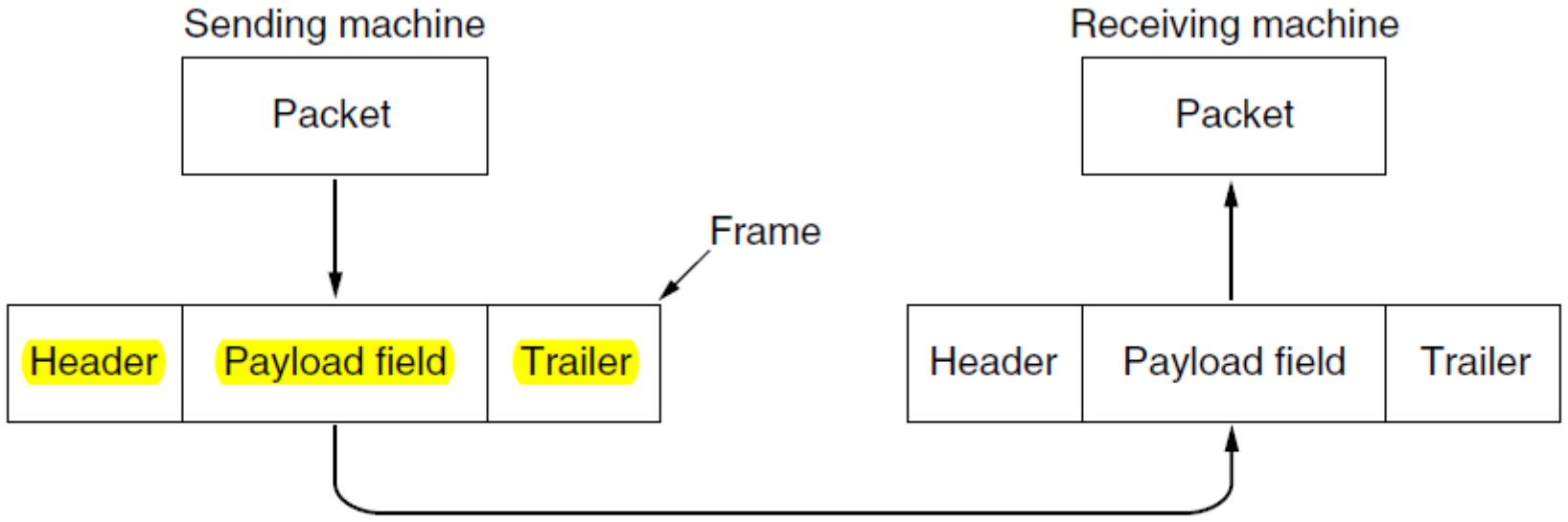


Figure 3-1. Relationship between packets and frames.

Data Link Layer: Framing

- The physical layer is to accept a raw bit stream and to attempt to deliver it to the destination.
- But the channel is **noisy**, so the physical layer will add some **redundancy** to its signals to reduce the bit error rate to a tolerable level.
- The usual approach for the data link layer is to break up the bit stream into discrete **frames**, compute a short token called a **checksum** for each frame, and include the checksum in the frame when it is transmitted.
- When a frame arrives at the destination, the checksum is recomputed. If the newly computed checksum is different from the one contained in the frame, the data link layer knows that an error has occurred and takes steps to deal with it (discard or send back an error report).

Data Link Layer: Framing

- It is difficult to break up the bit stream into frames, a good design must make it easy for a receiver to find **the start** of new frames while using little of the channel bandwidth.
- There are about **four** methods.
  - 1) Byte count
  - 2) Flag bytes with byte stuffing
  - 3) Flag bits with bit stuffing
  - 4) Physical layer coding violations

Framing — **Byte Count**

- To use a field in the header to specify the number of bytes in the frame.

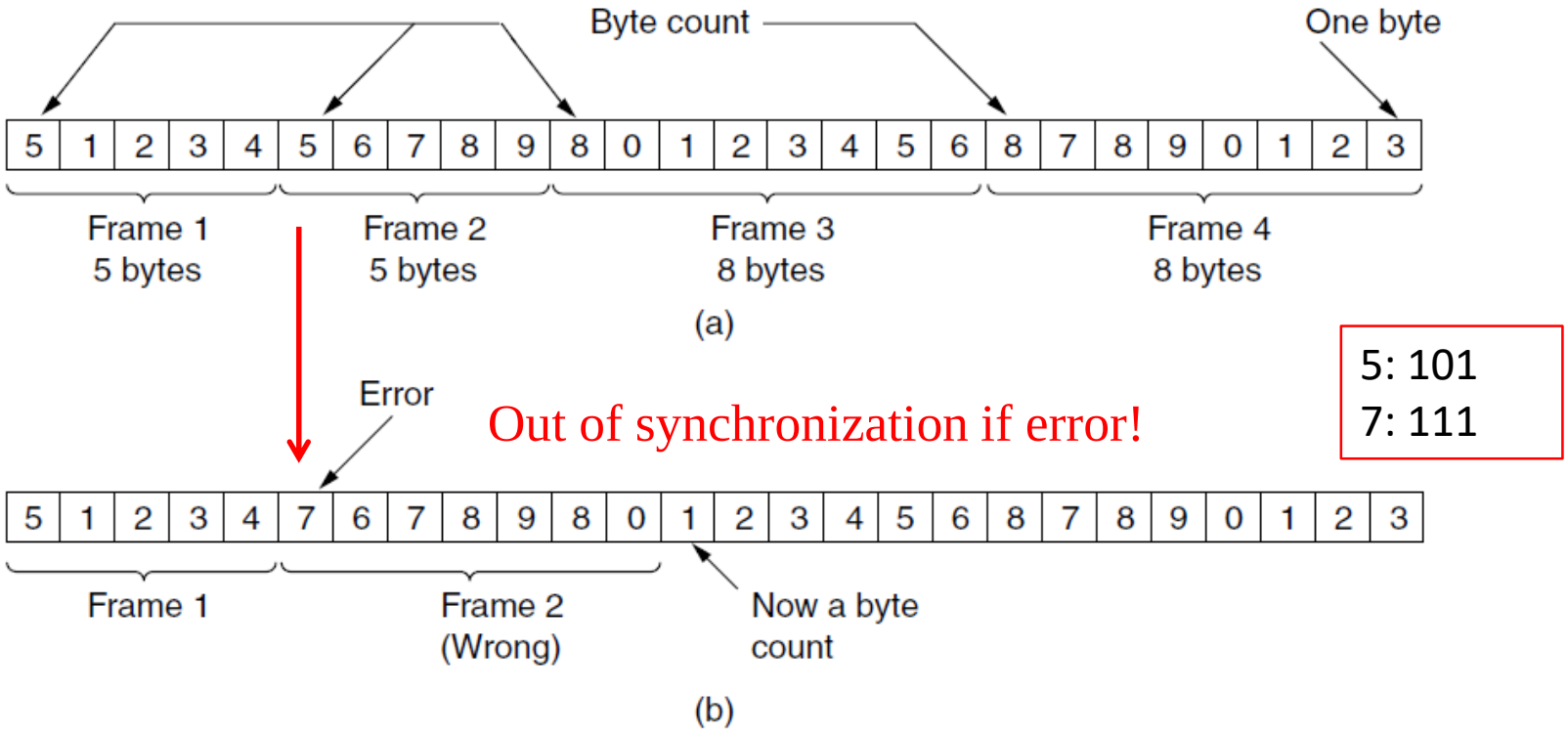
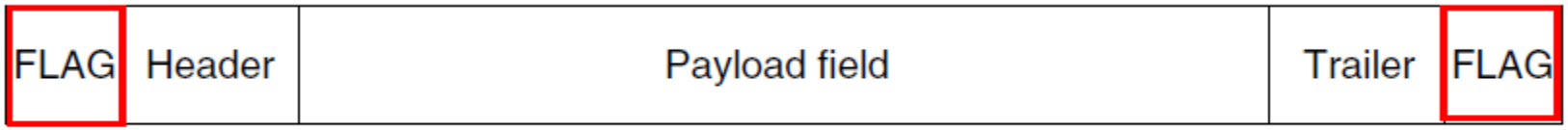


Figure 3-3. A byte stream. (a) Without errors. (b) With one error.

Framing — Flag Byte with **Byte Stuffing** (I)

- The second framing method gets around the problem of resynchronization after an error by having each frame start and end with **special bytes**.
  - Often the same byte, called **a flag byte**, is used as both the starting and ending delimiter.
  - Two consecutive flag bytes indicate the end of one frame and the start of the next.
    - If the receiver ever loses synchronization it can just search for two flag bytes to find the end of current frame and the start of the next frame.



(a)

Framing — Flag Byte with **Byte Stuffing** (II)

- But it may happen that the flag byte occurs in the data, especially when binary data such as photographs or songs are being transmitted.
  - In such a case, the flag bytes occurred in the data will be interference signal.
  - Solution: to insert **a special escape byte** (ESC) just before each "accidental" flag byte in the data.
  - The next question is: what happens if the an escape byte occurs in the middle of data?
  - Solution: to insert an escape byte.

- Application Example: **PPP** (Point-to-Point Protocol)

Framing — Flag Byte with **Byte Stuffing** (III)

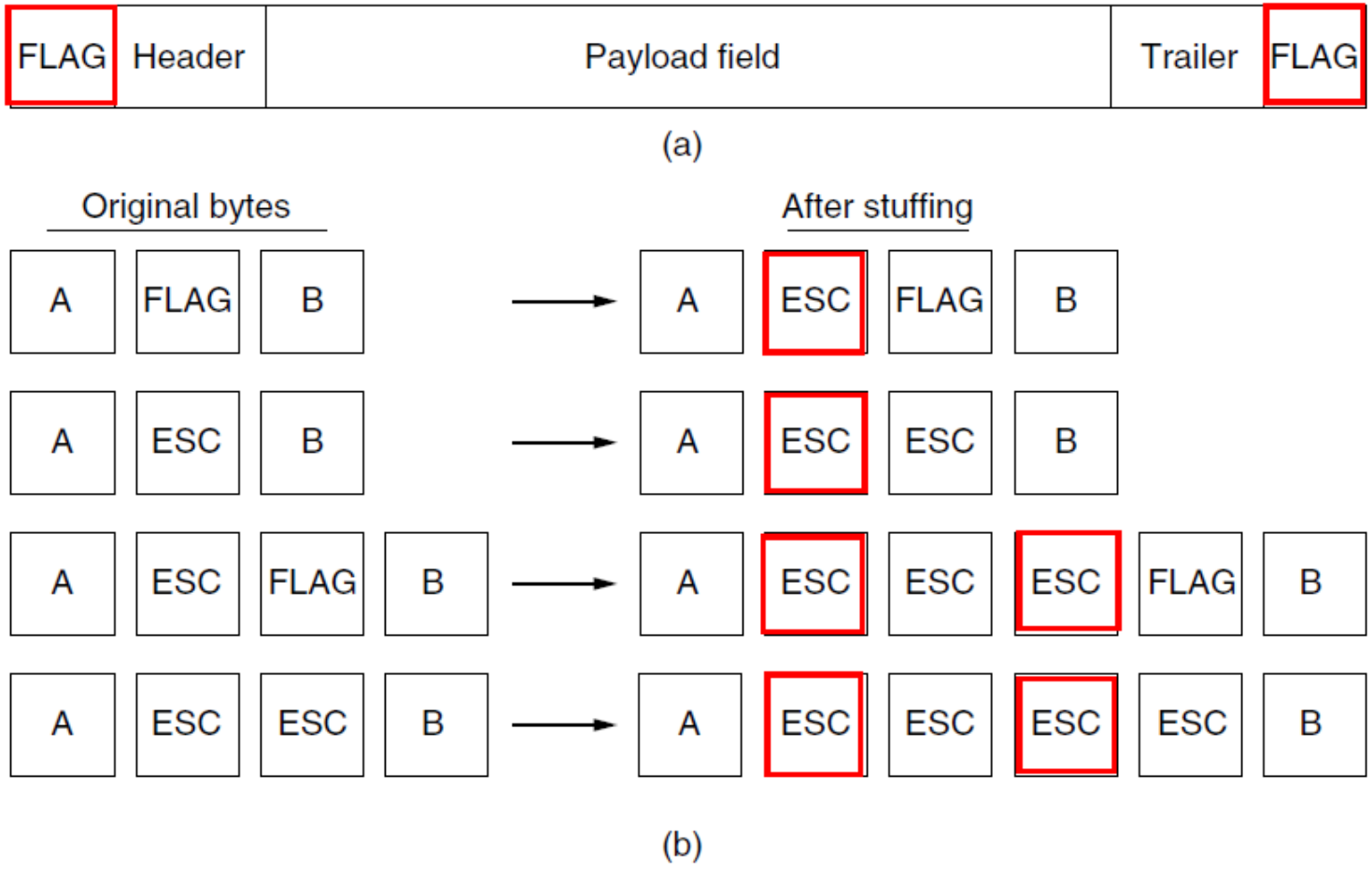


Figure 3-4. (a) A frame delimited by flag bytes. (b) Four examples of byte sequences before and after byte stuffing.

Framing — Flag Byte with **Bit Stuffing** (I)

- Framing can also be done at the bit level
- Example: **HDLC** (High-Level Data Link Control) protocol
  - Each frame begins and ends with a special bit pattern 01111110 or 0x7E in hexadecimal. This pattern is a flag byte.
  - Whenever the sender's data link layer encounters five consecutive 1s in the *data*, it automatically stuffs a 0 bit into the outgoing bit stream.
- The advantage of bit stuffing is that it ensures a minimum density of transition that help the physical layer maintain synchronization.
- Application Example: **USB** (Universal Serial Bus) uses bit stuffing.

Framing — Flag Byte with **Bit Stuffing** (II)

- When the receiver sees five consecutive incoming 1 bits, followed by a 0 bit, it automatically destuffs (i.e. deletes) the 0 bit.

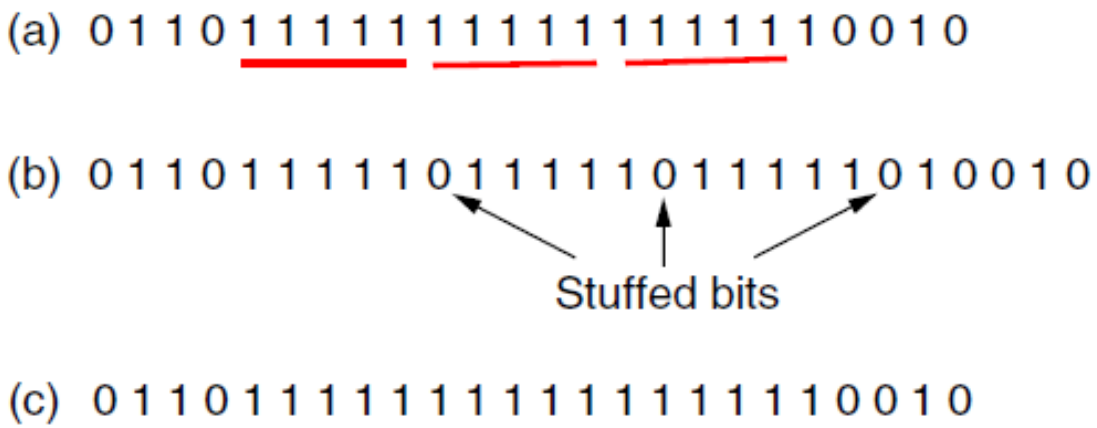


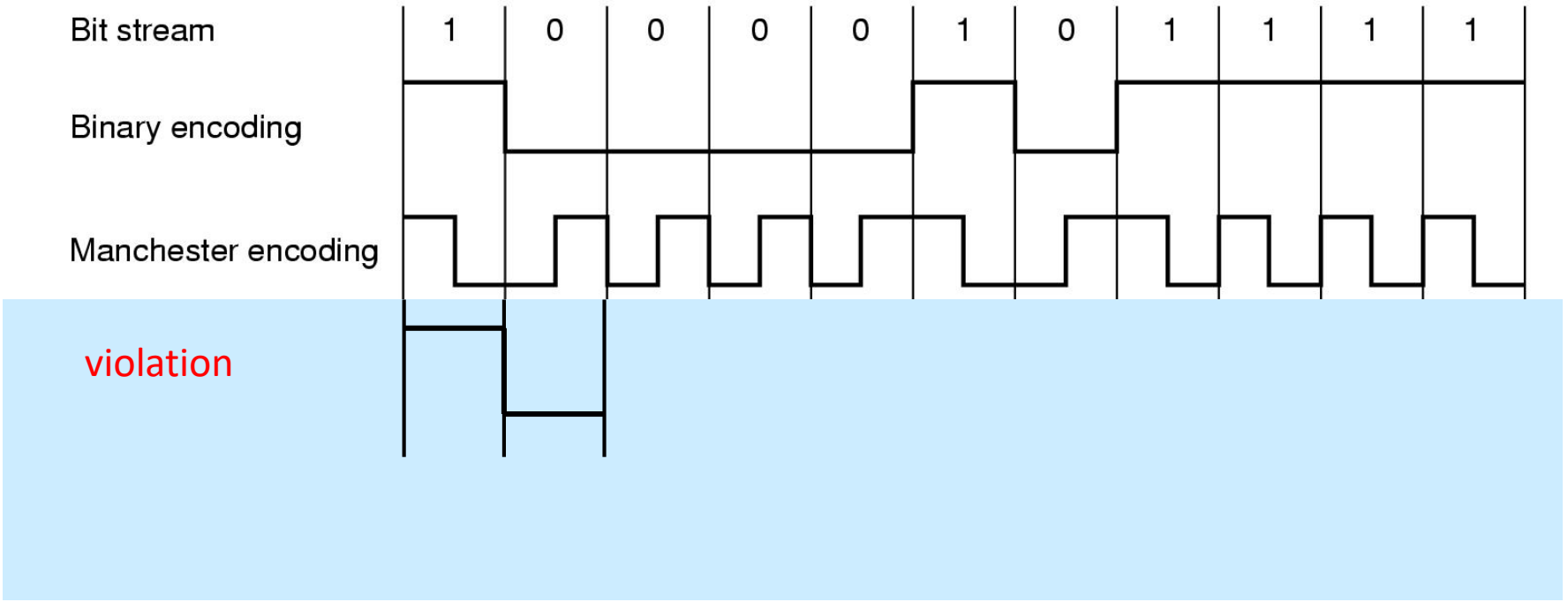
Figure 3-5. Bit stuffing. (a) The original data. (b) The data as they appear on the line. (c) The data as they are stored in the receiver's memory after destuffing.

Framing — **Coding Violations** (I)

- With both bit and byte stuffing, a side effect is that the length of a frame now depends on the contents of the data it carries.
- At the physical layer, the encoding of bits as signals often includes **redundancy** to help the receiver.
- Example: in the 4B/5B line code 4 data bits are mapped to 5 signal bits, and this means that 16 out of the 32 signal possibilities are not used. We can use some reserved signals to indicate the start and the end frames.

- Many data link protocols use a combination of these methods.
  - Example: Ethernet and 802.11 is to have a frame begin with a well-defined pattern called a preamble. This pattern might be quite long (72 bits is typical for 802.11) to allow the receiver to prepare for an incoming packet. The preamble is then followed by a length (i.e. count) field in the header that is used to locate the end of the frame.

Framing — **Coding Violations** (II)



♠ Use no transition in a slot (= coding violation) as start of frame.

Outline

- Overview of Data link layer
- Data link layer: **Framing**
- Data link layer: **Error Control**
  - **Error Correction**
  - **Error Detection**
- Elementary data link protocols
- **Sliding window protocols**
- Examples of data link protocols

Outline

- Overview of Data link layer
- Data link layer: **Framing**
- Data link layer: **Error Control**
  - **Error Correction**
  - **Error Detection**
- Elementary data link protocols
- **Sliding window protocols**
- Examples of data link protocols

Data Link Layer: Error Control

- Network designers have developed two basic strategies for dealing with errors. Both add **redundant** information to the data that is sent.
  - One strategy is to include enough redundant information to enable the receiver to deduce what the transmitted data must have been. — **Error-Correcting Codes**
    - For **unreliable** channels, such as wireless links, it is better to add redundancy to each block so that the receiver is able to figure out what the originally transmitted block was.
  - The other is to include only enough redundancy to allow the receiver to deduce that an error has occurred (but not which error) and have it request a **retransmission**. — **Error-Detecting Codes** (Forward Error Correction, FEC)
    - For **highly reliable** channels, such as optical fiber, it is cheaper to use an error-detecting code.
- **Neither** error-correcting codes **nor** error-detecting codes can handle *all possible errors* since the redundant bits that offer protection are as likely to be received in error as the data bits.

Error Models of Communication Channels

- Errors are caused by extreme values of **the thermal noise** that overwhelm the signal briefly and occasionally, giving rise to *isolated single-bit errors*.
- Another model is that errors tend to **come in bursts** rather than singly.
  - Deep fade on a wireless channel or transient electrical interference on a wired channel.
- **Either** the error-correcting codes **or** the error-detecting codes are **not only used** in the data link layer, but are widely used in other layers as well because reliability is an overall concern.
  - Error-correcting codes are seen in the physical layer, particularly for noisy channels, and in higher layers, particular for real-time media and content distribution.
  - Error-detecting codes are commonly used in link, network, and transport layers.

Error Control — **Error-Correcting**

- We will discuss two different error-correcting codes
  - Hamming codes
  - Binary convolutional codes
  - Reed-Solomon codes
  - Low-density parity check codes

- Block code — a frame consists of *m* data (or message) bits and *r* redundant (or check) bits. The *r* check bits are computed solely as a function of the *m* data bits with which they are associated.
  - In **a systematic code**, the *m* data bits are sent directly, along with the check bits, rather than being encoded themselves before they are sent.
  - In **a linear code**, the *r* check bits are computed as a linear function of the *m* data bits.

Error Control — **Error-Correcting**

- **Hamming distance** (Hamming, 1950) — The number of bit positions in which two codewords differ.
  - Example 10001001, 10110001
  - Its significance is that if two codewords are a Hamming distance *d* apart, it will require *d* single-bit errors to convert one into the other.

```
10001001
10110001
--------
00111000
```

- Given the algorithm for computing the check bits, it is possible to construct a complete list of the legal codewords, and from this list to find the two codewords with the **smallest** Hamming distance. This distance is **the Hamming distance of the complete code**.

- Principle — Let the total length of a block be *n* (i.e. $n = m + r$). An *n*-bit unit containing data and check bits is referred to as an *n*-bit **codeword**.
  - **The code rate** is the fraction of the codeword that carries information that is not redundant, or *m*/*n*.
    - They might be ½ for a noisy channel, in which case half of the received information is redundant.
    - Or close to 1 for a high-quality channel, with only a small number of check bits added to a large message.

Error Control — **Error-Correcting**

- The error-detecting and error-correcting properties of a block code depend on its Hamming distance.
  - To reliably detect *d* errors, you need a distance *d* + 1 code
  - To correct *d* errors, you need a distance 2*d* + 1 code.
  - Example — consider a code with only four valid codewords: 0000000000, 0000011111, 1111100000, 1111111111. This code has a distance of 5, which means that it can correct double errors or detect quadruple errors.
    - If a codeword 0000000111 arrives and we expect only single- or double-bit errors, the receiver will know that the original must have been 0000011111.
    - However, a triple error changes 0000000000 into 0000000111, the error will not be corrected properly (since 0000011111 has the smaller Hamming distances (2) to the arrived codeword, while the Hamming distance between 0000000000 and 0000000111 is 3).

Error Control — **Error-Correcting**

- 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. 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

$$(m + r + 1) \leq 2^r$$

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

Error Control — **Error-Correcting**

- **Hamming codes**
  - In the most general case where all codewords need to be evaluated as candidates, this task can be a time-consuming search.
  - Instead, practical codes are designed so that they admit shortcuts to find what was likely the original codeword.
  - In a 7-bit message, there are seven possible single bit errors, so three error control bits could potentially specify not only that an error occurred but also which bit caused the error.

$$(m + r + 1) \leq 2^r$$

In Hamming codes, $m = 4$, and $r = 3$.

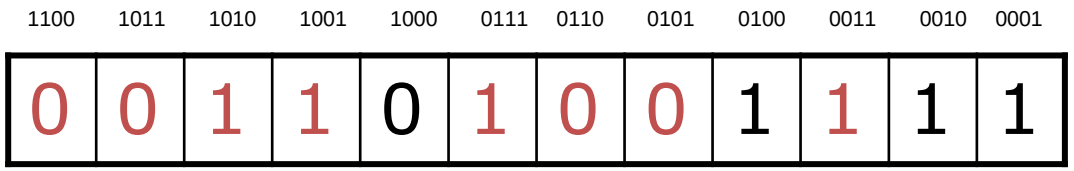
Error Control: **Hamming Code**

- Encoding: k data bits + $(n-k)$ check bits
  - Counting from the least-significant position, the Hamming check bits are inserted at positions that are a power of 2.
  - To calculate the check bits, each data position which has a value 1 is represented by a binary value equal to its position. All of the position values are then XORed together to produce the bits of the Hamming code.

- Decoding: compares received $(n-k)$ bits with calculated $(n-k)$ bits using XOR
  - Resulting $(n-k)$ bits called *syndrome word*
  - Syndrome range is between 0 and $2^{(n-k)} - 1$
  - Each bit of syndrome indicates a match (0) or conflict (1) in that bit position

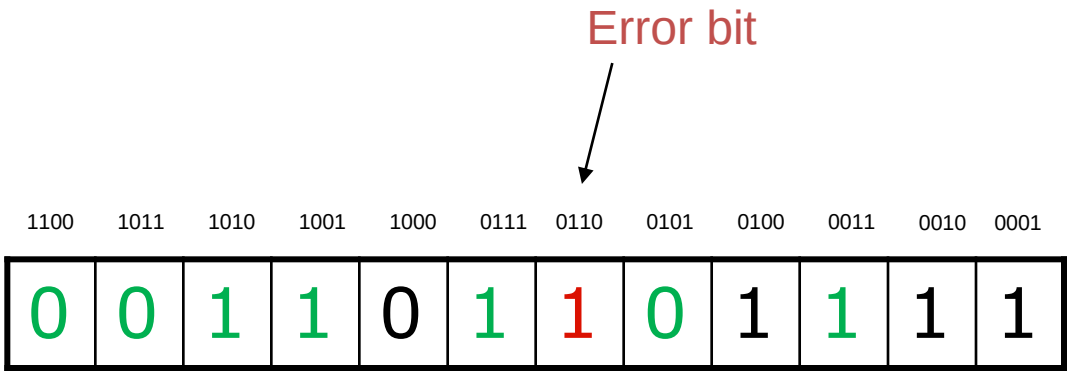
An Example

- **Transmitted**
- A 8-bit data message is 00111001
- 1010⊕1001⊕0111⊕0011
  = 0111 (this is the Hamming code)



- The transmitted block is 001101001111

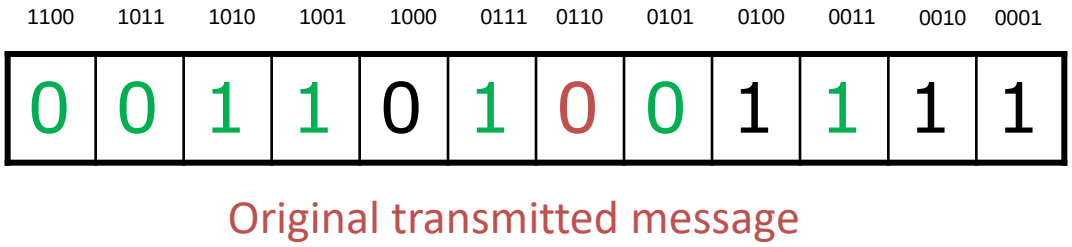
An Example



- **Received**
- The Hamming code is 0111.
- Syndrome code is

  0111⊕1010⊕1001⊕0111⊕0110⊕0011

  =0110 (the position where the bit is error)

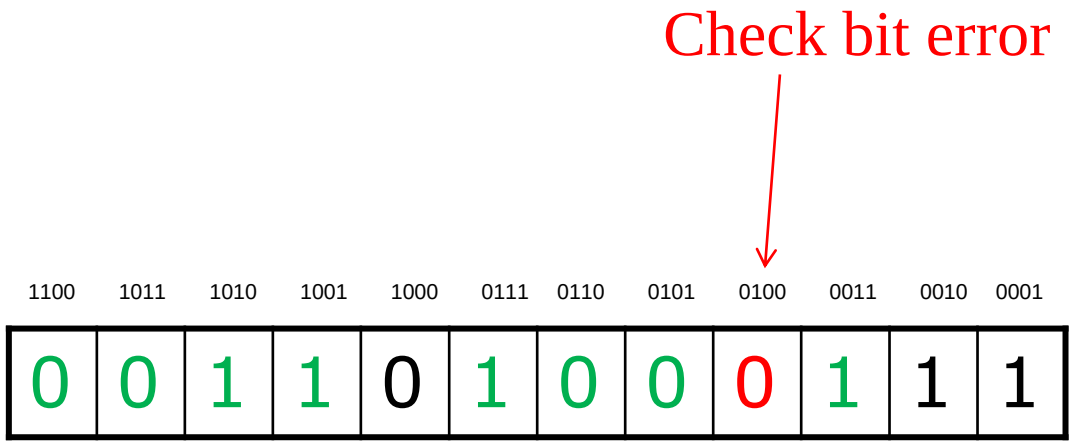


Characteristics of Syndrome

- If the syndrome contains all 0s, no error has been detected.
- If the syndrome contains one and only one bit set to 1, then an error has occurred in one of the check bits. No correction is needed.
- If the syndrome contains more than one bit set to 1, then the numerical value of the syndrome indicates the position of the data bit in error. This data bit is inverted for correction.

Example for Error in Check Bit

- Received message
- 1010⊕1001⊕0111⊕0011⊕0010⊕0001=0100



Error Control — **Convolutional Code**

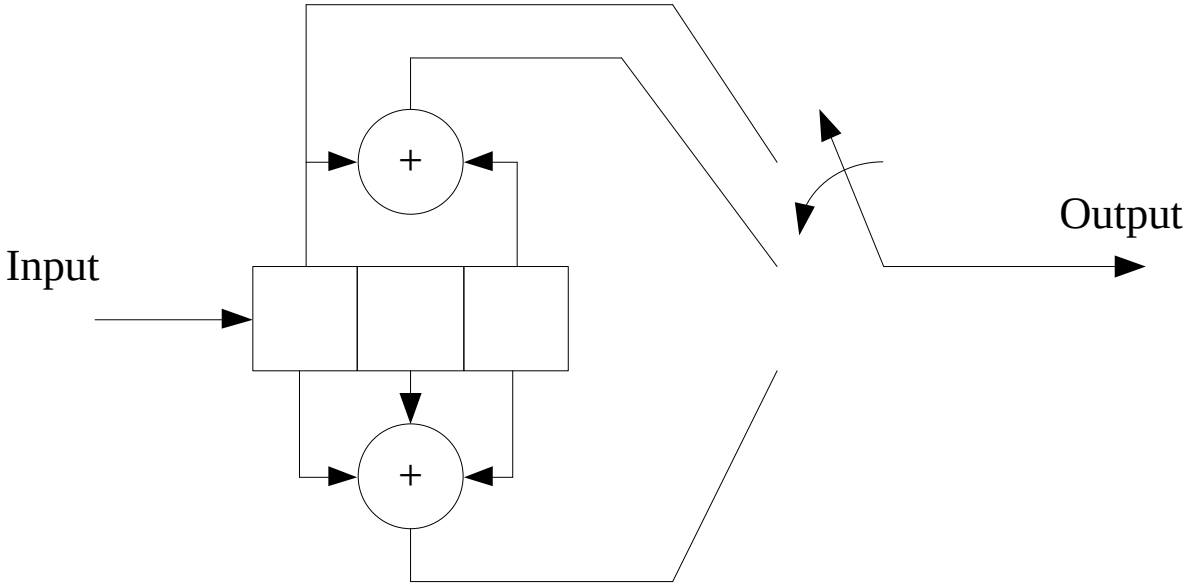
- The convolutional code is **not** a block code.
  - The convolutional code maps the *m*-bit input (data or message) into an *n*-bit codeword that the original *m* bits do not appear in the codeword.
  - The output depends on the current and previous input bits. — The convolutional code has **memory**.
  - The number of previous bits on which the output depends is called the constraint length of the code (*K*).
  - (*n*, *m*, *K*) code

Error Control — **Convolutional Code**

- (n, m, K) code
  - Input processes m bits at a time
  - Output produces n bits for every m input bits
  - K = constraint length (factor)
  - m and n generally very small
- n-bit output of (n, m, K) code depends on
  - Current block of m input bits
  - Previous K–1 blocks of m input bits

Example: *K*=3, *m*=1, *n*=3 Convolutional Encoder (I)

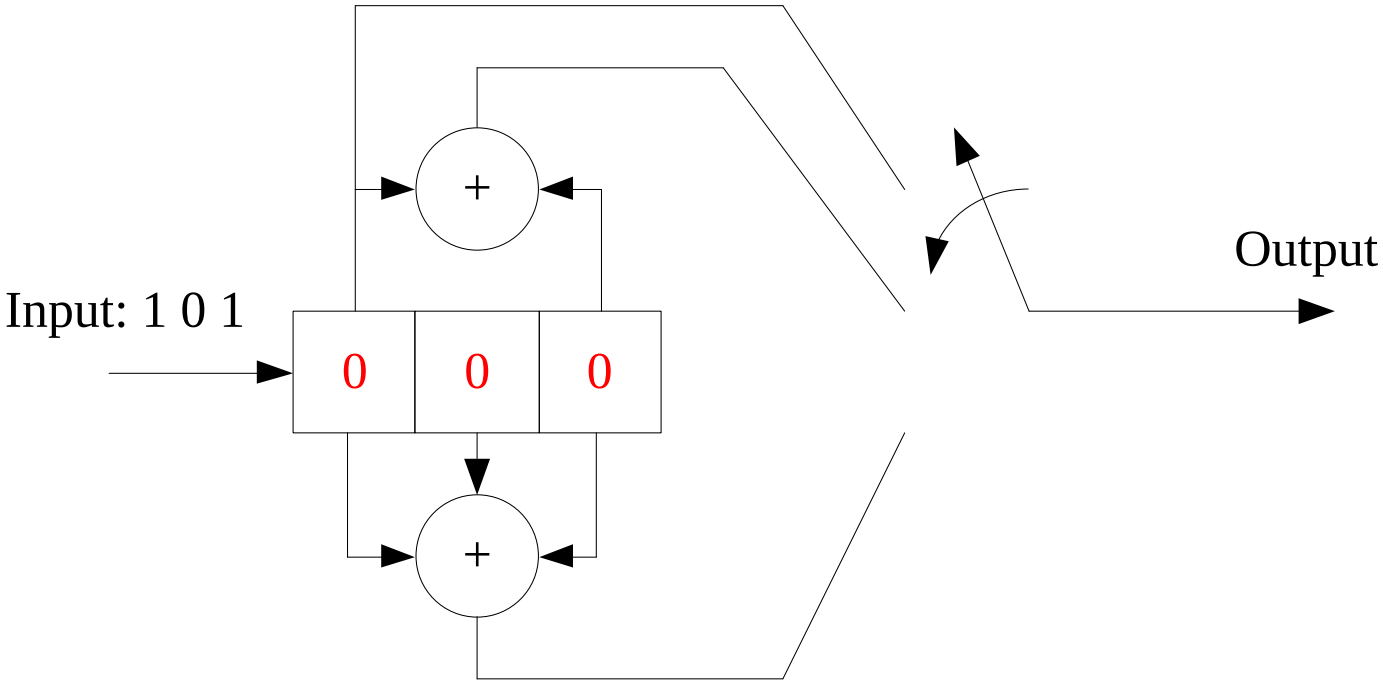
K = 3, m = 1, n = 3 Convolutional Encoder



Initially the shift register is assumed to be in the all-zero state

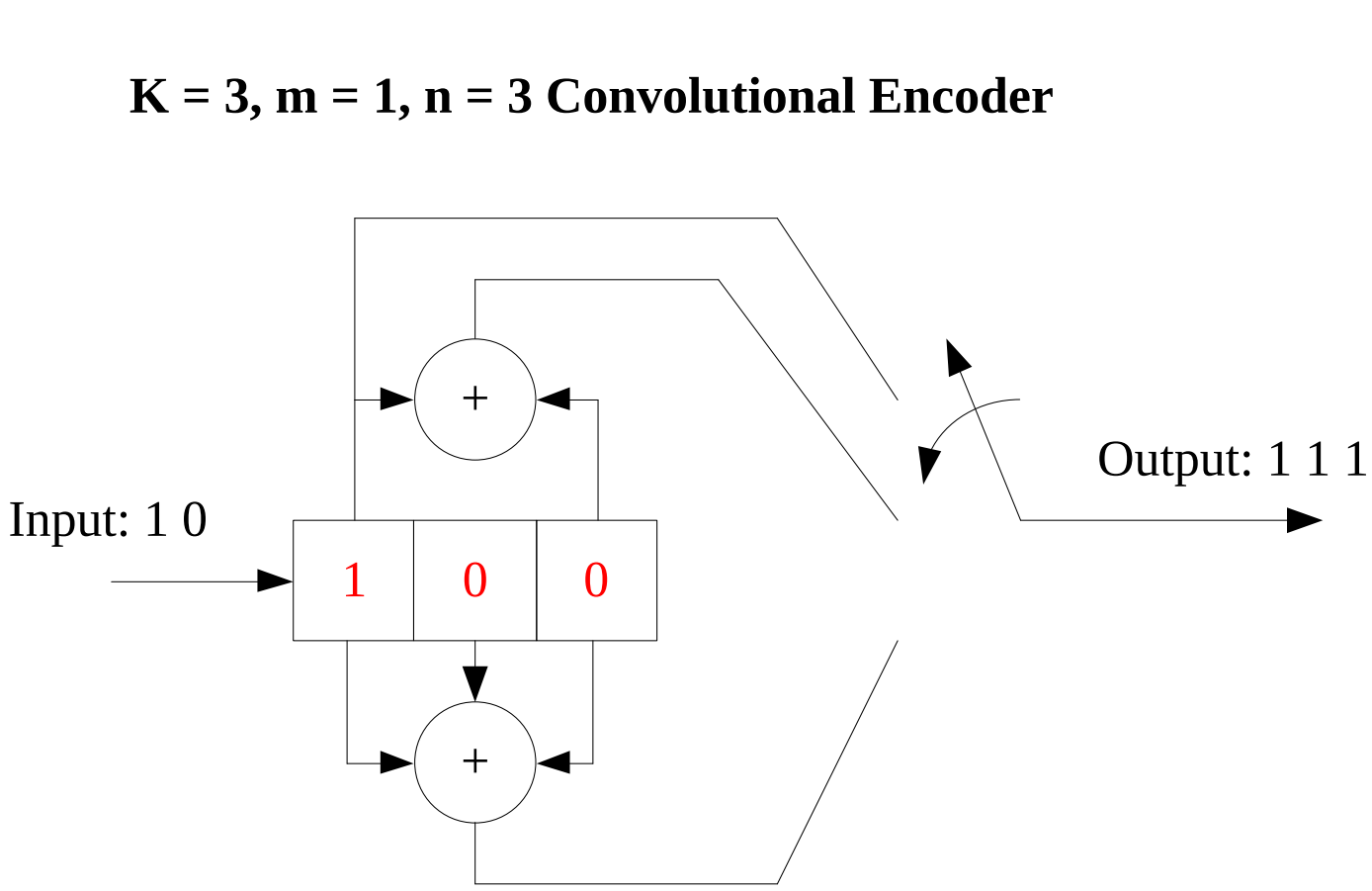
Example: *K*=3, *m*=1, *n*=3 Convolutional Encoder (II)

K = 3, m = 1, n = 3 Convolutional Encoder



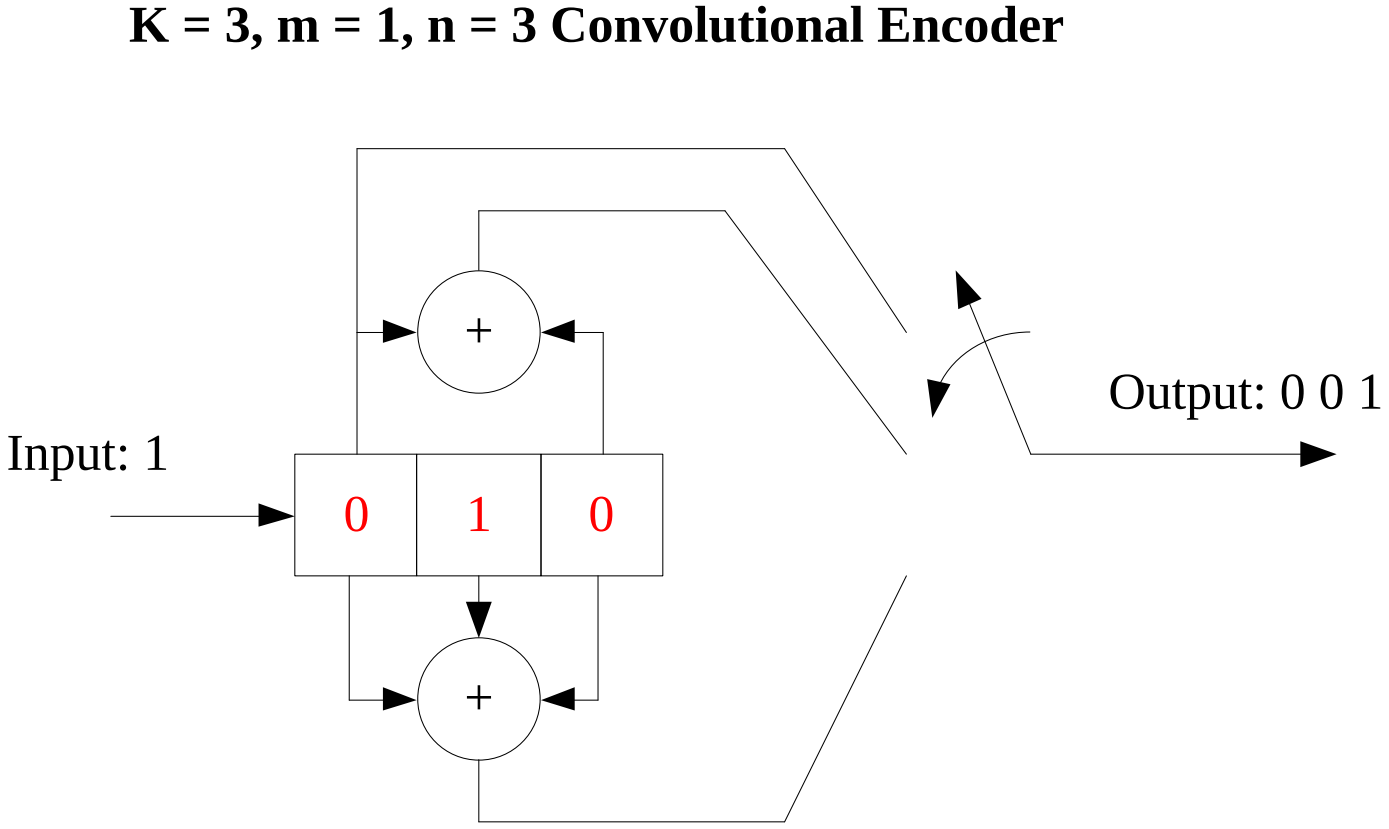
Initially the shift register is assumed to be in the all-zero state

Example: *K*=3, *m*=1, *n*=3 Convolutional Encoder (III)



Initially the shift register is assumed to be in the all-zero state

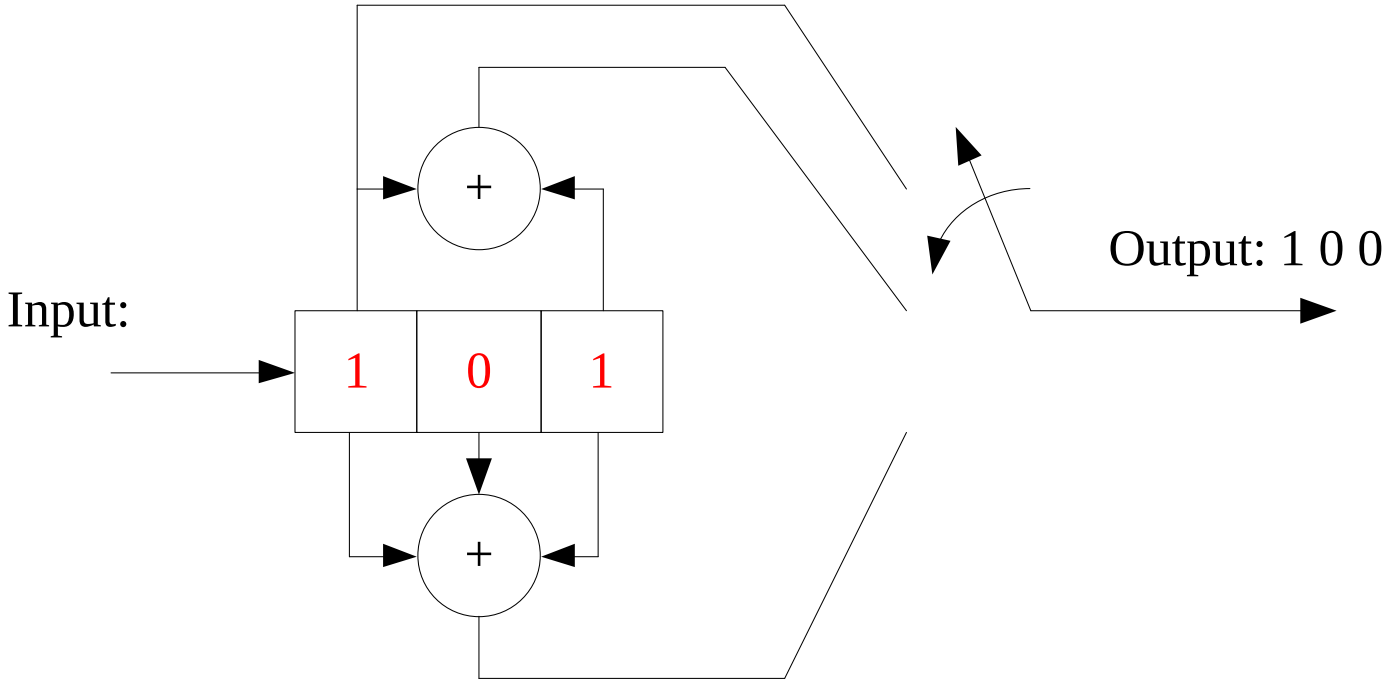
Example: *K*=3, *m*=1, *n*=3 Convolutional Encoder (IV)



Initially the shift register is assumed to be in the all-zero state

Example: *K*=3, *m*=1, *n*=3 Convolutional Encoder (V)

K = 3, m = 1, n = 3 Convolutional Encoder



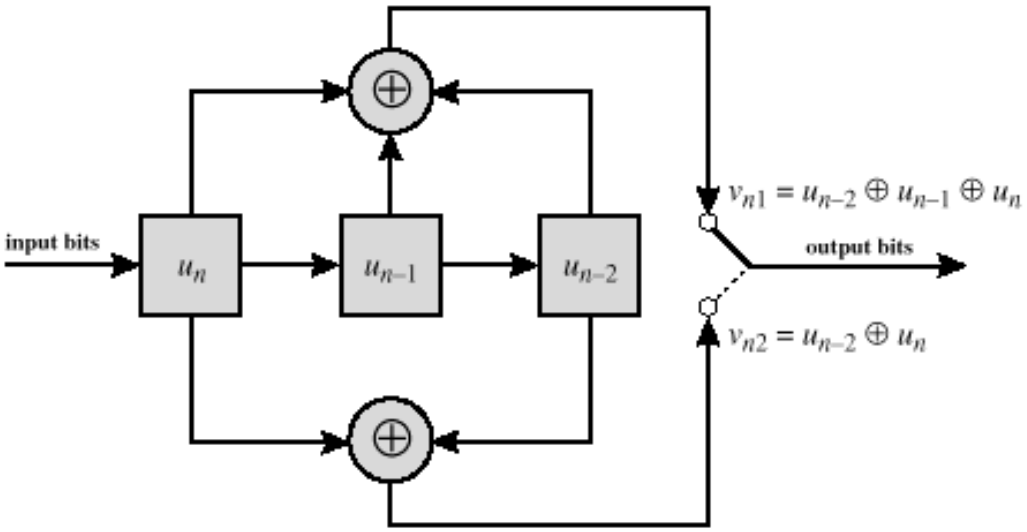
Initially the shift register is assumed to be in the all-zero state

Convolutional Decoder *

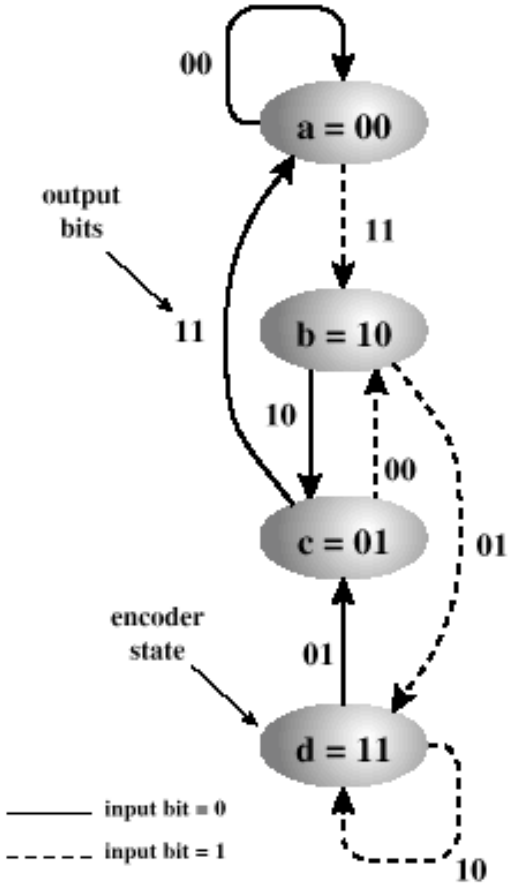
- A convolutional code is decoded by finding the sequence of input bits that is most likely to have produced the observed sequence of output bits.
  - **The Viterbi algorithm** (1973) [2]: The algorithm walks the observed sequence, keeping for each step and for each possible internal state the input sequence that would have produced the observed sequence with the **fewest** errors. The input sequence requiring the fewest errors at the end is the most likely message.

Convolutional Encoder with *

(n, m, K)=(2, 1, 3)



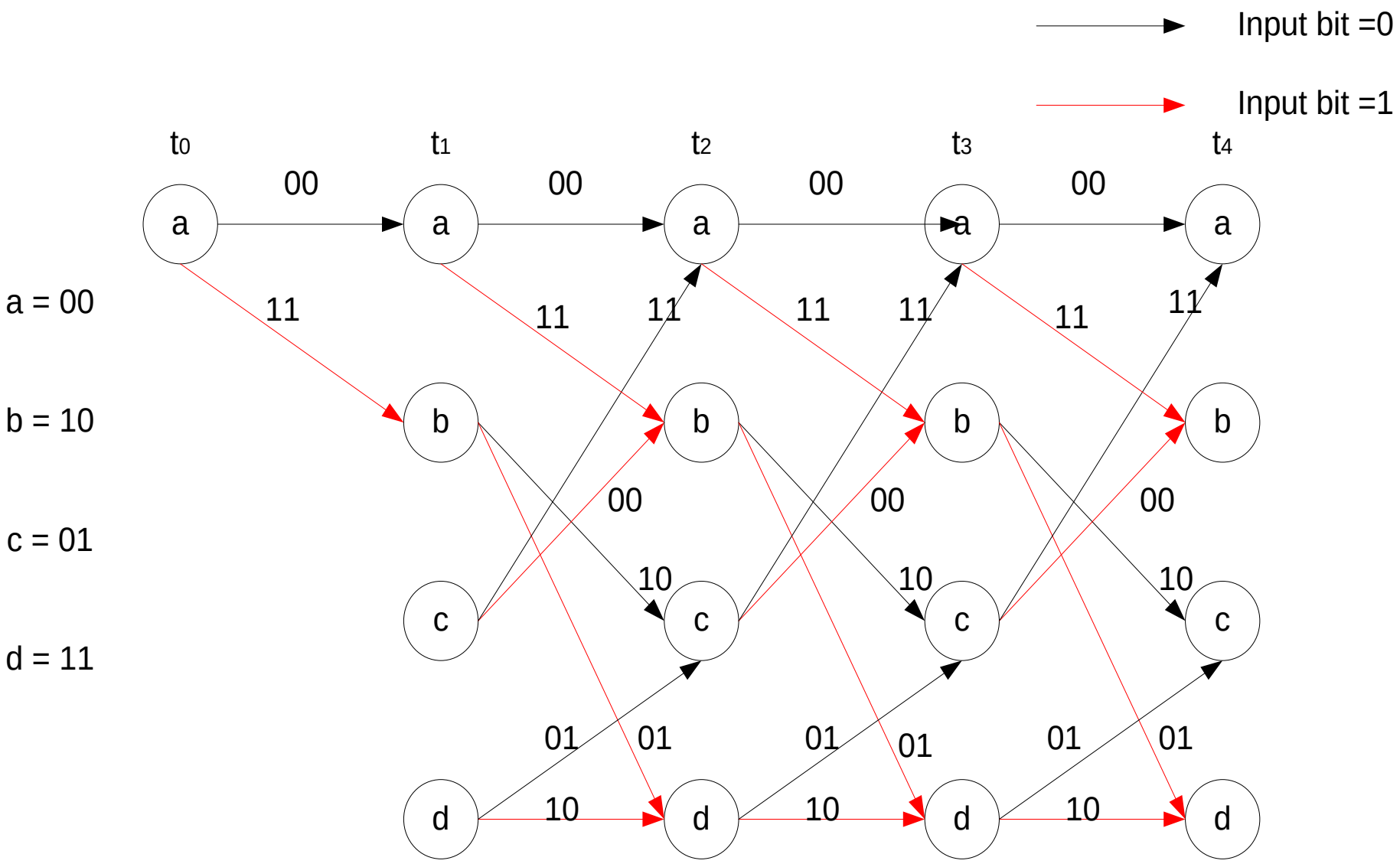
(a) Encoder shift register



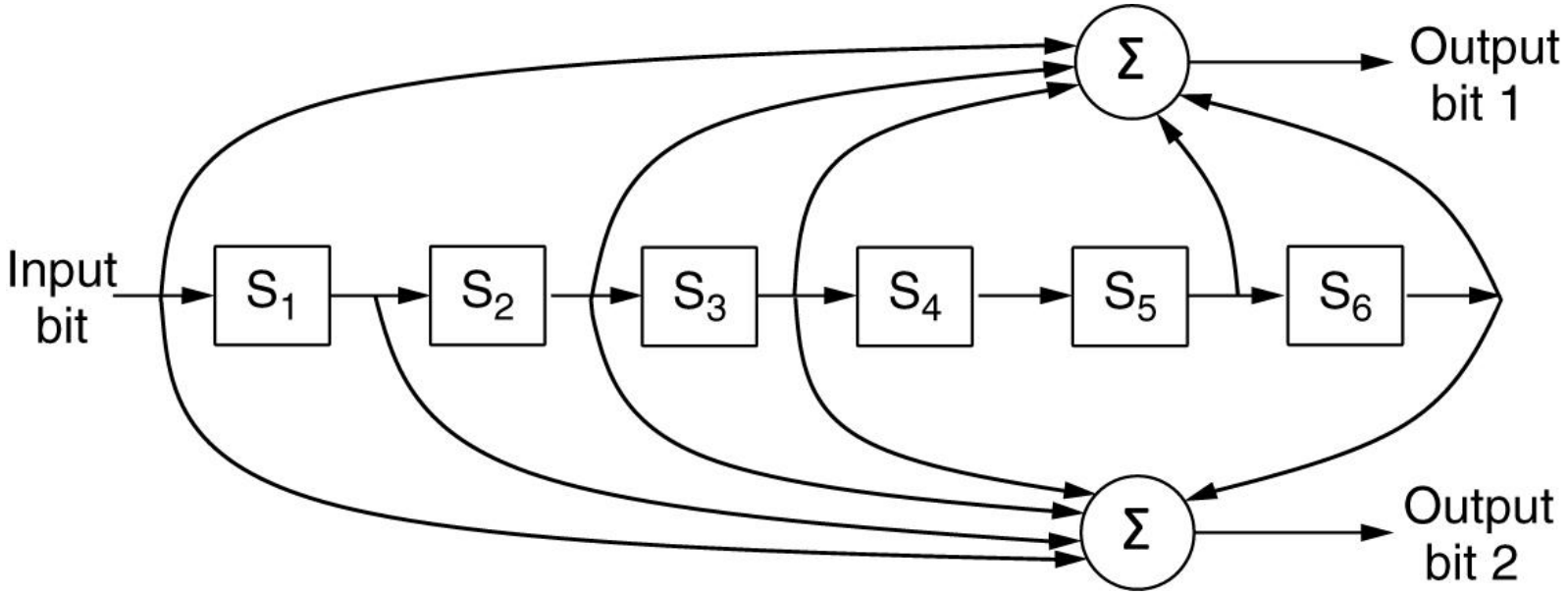
(b) Encoder state diagram

Convolution Encoder with (n, m, K) = (2, 1, 3)

Trellis Diagram [2]*



The NASA binary convolutional code used in 802.11



Error Correcting Codes

- The Reed-Solomon Code (Massey, 1969)
  - Unlike Hamming codes, which operate on individual bits, Reed-Solomon codes operate on *m* bit symbols.
  - Reed-Solomon codes are based on the fact that every *n* degree polynomial is uniquely determined by *n*+1 points.
  - Reed-Solomon codes are widely used in practice, particular for burst errors.

- The LDPC (Low-Density Parity Check) Code (Gallagher, 1962)
  - In an LDPC, each output bit is formed from only a fraction of the input bits.
    - This leads to a matrix representation of the code that has a low density of 1s, hence the name for the code.

Error Control — **Error-Detecting**

- Error-correcting codes are widely used on wireless links, which are notoriously noisy and error prone when compared to optical fiber.
  - Over fiber or high-quality copper, the error rate is much lower, so **error detection** and **retransmission** is usually more efficient there for dealing with *the occasional error*.
- We will examine three different error-detecting codes
  - Parity
  - Checksums
  - Cyclic Redundancy Checks (CRCs)

Error Detecting: **Parity**

- A single parity bit is appended to the data. The parity is chosen so that the number of 1 bits in the codeword is even or odd.
- For example, when 1011010 is sent in even parity, a bit is added to the end to make it 10110100. With odd parity 1011010 becomes 10110101.
- A code with a single parity bit has a distance of 2, since any single bit error produces a codeword with the wrong parity.
  - It can detect **single-bit errors**.

Analysis of the Advantage of Parity

- Example: Typical LAN links provide bit error rates of 10^{-10}. Let the block size be 1000 bits.
- 1) To provide **error correction** for 1000-bit blocks,
  - Based on $(m + r + 1) \leq 2^r \rightarrow r = 10$ that 10 check bits are needed
  - A megabit of data would require 10,000 check bits.
- 2) To provide **error detecting** for 1000-bit blocks
  - One parity bit per block will suffice.
  - Once *every 1000 blocks*, a block will be found to be in error and an extra block (1001 bits) will have to be transmitted to repair the error.
  - Only 2001 bits per megabit of data versus 10,000 bits for a Hamming code. (1000 (每兆比特数据总共需要的校验位) + 1001 (错误重发))

Analysis of the Difficulty of Parity

- If the block is badly garbled by **a long burst error**, the probability that the error will be detected is only 0.5, which is hardly acceptable.
- However, there is something else we can do that provides better protection against **burst errors**: we can compute the parity bits over the data in a different order than the order in which the data bits are transmitted.
  - **Interleaving**
  - **Two-dimensional parity** [4] — a simple generalization of one-bit parity

Interleaving

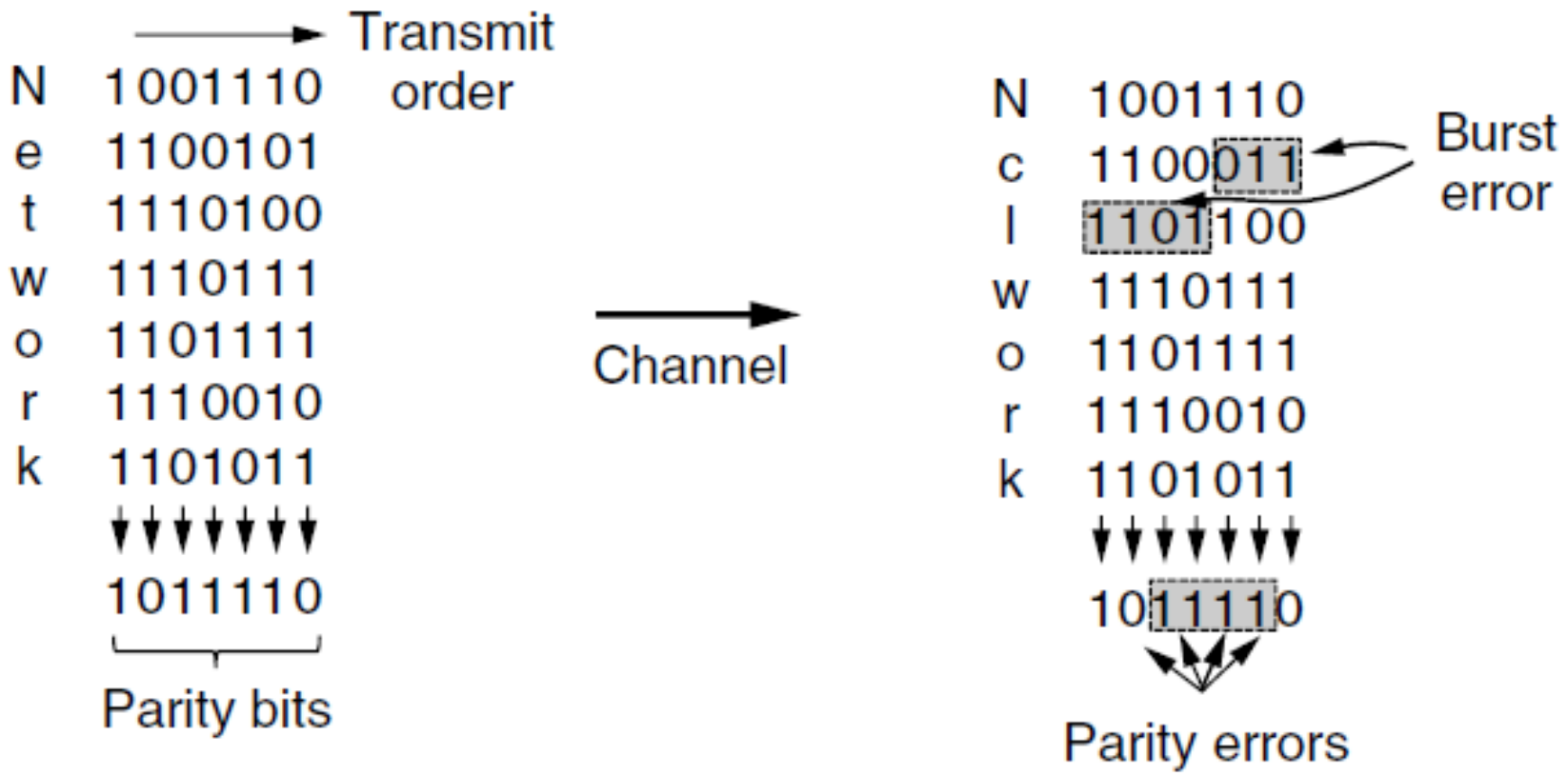
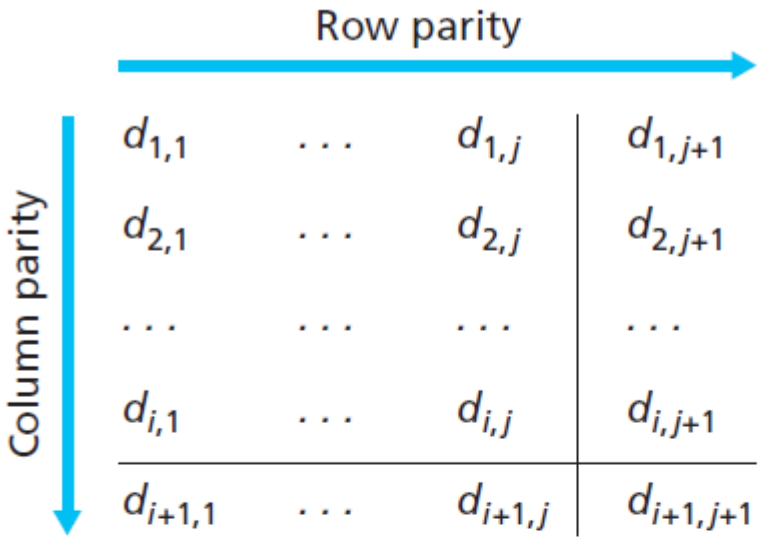
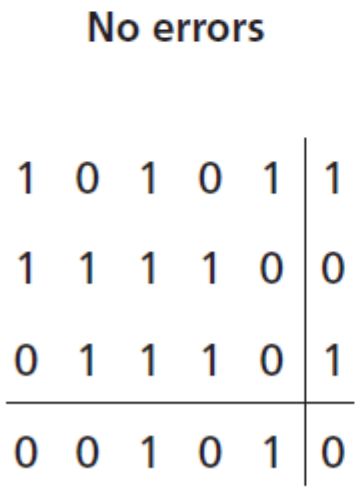
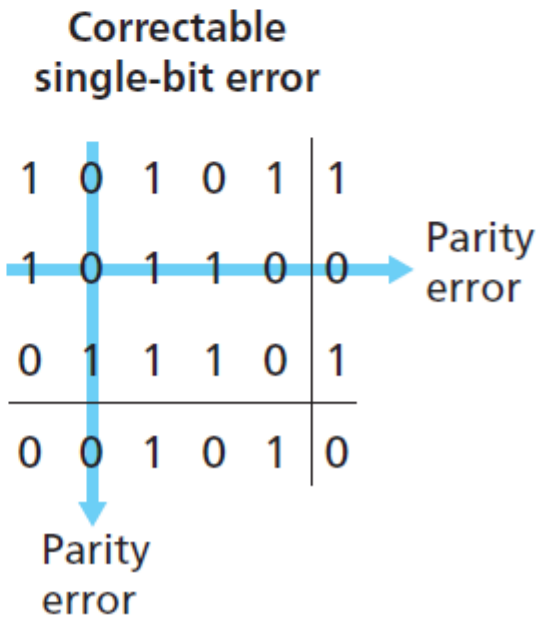


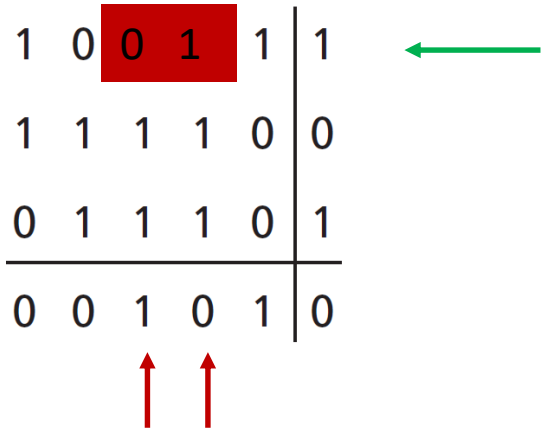
Figure 3-8. Interleaving of parity bits to detect a burst error.

An Two Dimensional Even Parity Example (Figure 5.6 in [4])









Two-dimensional parity can detect (but not correct!) any combination of two errors in a packet.

Error Detecting: **Checksum**

- Idea: sum up data in N-bit words
  - Widely used in, e.g., TCP/IP/UDP

- Stronger protection than parity

The 16-bit Internet Checksum (I)

- The Internet checksum is computed in one's complement arithmetic instead of the modulo 2^{16} sum.
  - In **one's complement arithmetic**, a negative number is the bitwise complement of its positive counterpart.
  - One's complement has two representations of zero, all 0s and all 1s.
  - Modern computers run **two's complement arithmetic**, in which a negative number is the one's complement plus one.
  - Example: if 001 represents "1", then 110 will represent "–1" in one's complement arithmetic, but 111 is the "–1" in two's complement arithmetic.

- The Internet checksum (RFC1071)

The 16-bit Internet Checksum (II)

- On **a two's complement** computer, the one's complement sum is equivalent to taking the sum modulo 2^{16} and adding any **overflow** of the high order bits back into the low-order bits.
- Sending:
- 1) Arrange data in 16-bit words
- 2) Put zero in checksum position, add
- 3) Add any carryover back to get 16 bits
- 4) Negative (complement) to get sum

0001
F203
F4F5
F6F7
0000 --- checksum
2ddf0
ddf2 (0010 -→1101)
220d

The 16-bit Internet Checksum (III)

- Receiving:
  - 1) Arrange data in 16-bit words
  - 2) Checksum will be non-zero, add
  - 3) Add any carryover back to get 16 bits
  - 4) Negate the result and check it is 0.

```
0001
F203
F4F5
F6F7
220d
2FFFd
-------
 FFFd
    2
FFFF
```

- The Internet checksum is efficient and simple but provide weak protection in some cases precisely because it is a simple sum.
  - It does not detect the deletion or addition of zero data, nor swapping parts of the message.

Error Detecting: **Cyclic Redundancy Check (CRC)**

- Transmitter
  - For a *m*-bit block, transmitter generates an (*n*–*m*)-bit frame check sequence (FCS)
  - Resulting frame of *n* bits is exactly divisible by predetermined bit pattern — key (or the generator polynomial) (Both the high- and low-order bits of the generator must be 1)

- Receiver
  - Divides incoming frame by predetermined bit pattern
  - If no remainder, assumes no error

Error Detecting: **CRC**

- Parameters:
  - T = n-bit frame to be transmitted
  - D = m-bit block of data or the message; the first m bits of T
  - F = $(n - m)$-bit FCS; the last $(n - m)$ bits of T
  - P = pattern of $n–m+1$ bits; this is the predetermined divisor
  - Q = Quotient
  - R = Remainder

The pattern P is chosen to be one bit longer than the desired FCS.

T

P

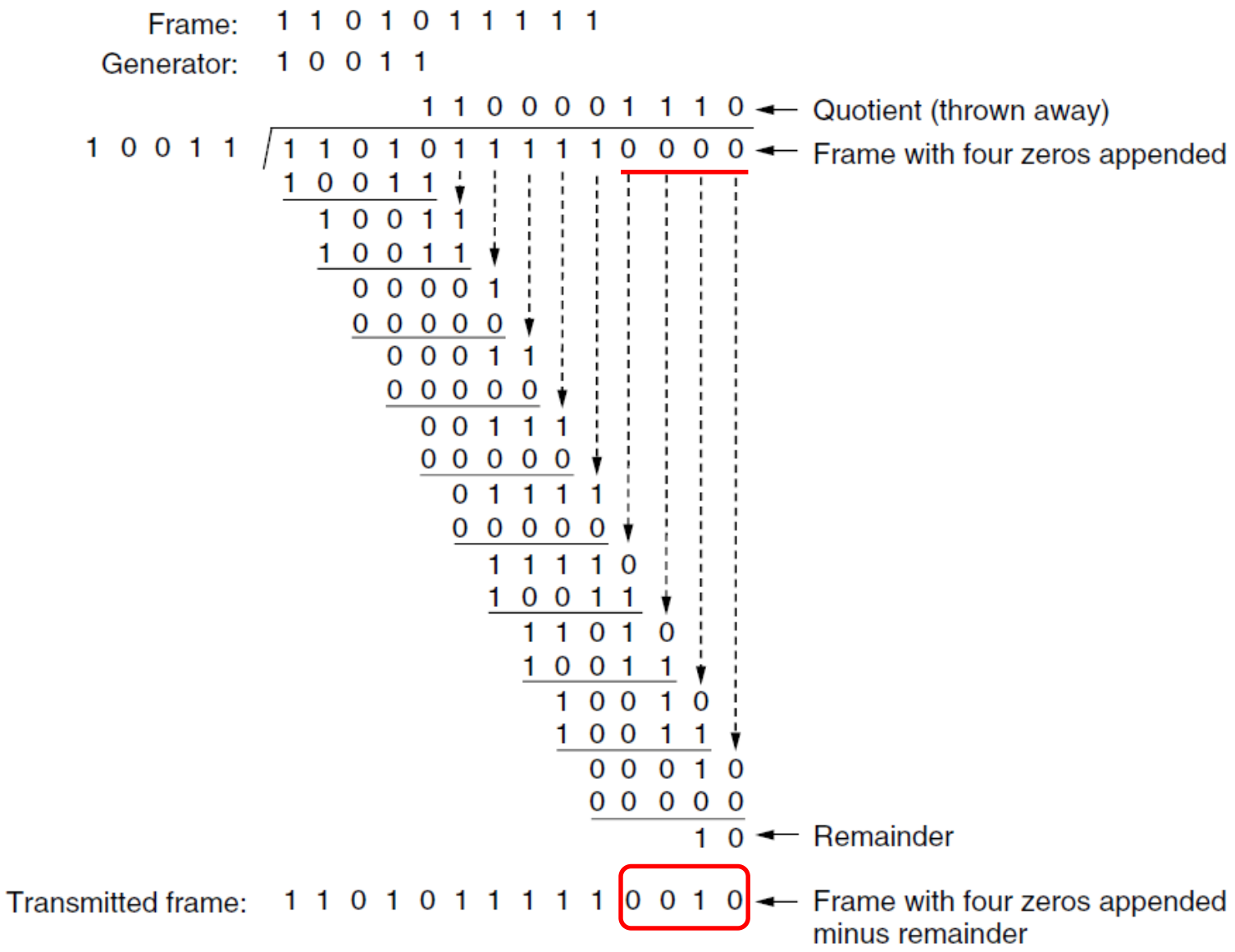


Figure 3-9. Example calculation of the CRC.

Think about if an error pattern is divisible by the generator P, what will happen?

Outline

- Overview of Data link layer
- Data link layer: **Framing**
- Data link layer: **Error Control**
  - **Error Correction**
  - **Error Detection**
- Elementary data link protocols
- **Sliding window protocols**
- Examples of data link protocols

Outline

- Overview of Data link layer
- Data link layer: **Framing**
- Data link layer: **Error Control**
  - **Error Correction**
  - **Error Detection**
- Elementary data link protocols
- **Sliding window protocols**
- Examples of data link protocols

A Utopian Simplex Protocol

- **Unrealistic** assumptions
  - 1) Data are transmitted in one direction only.
  - 2) Both the transmitting and receiving network layers are always ready.
  - 3) Processing time can be ignored.
  - 4) Infinite buffer space is available.
  - 5) The communication channel between the data link layer never damages or loses frames.

```
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.

A Simplex Stop-and-Wait Protocol for an **Error-Free** Channel

- This protocol is to tackle the problem of preventing the sender from flooding the receiver with frames faster than the latter is able to process them.
- A general solution to this problem is to have the receiver provide **feedback** to the sender.
  - After having passed a packet to its network layer, the receiver sends **a little dummy frame** (**an acknowledge frame**) back to the sender which, in effect, gives the sender permission to transmit the next frame.
  - This is a **stop-and-wait** protocol.
  - A **half-duplex** physical channel would suffice here.

```
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.

A Simplex Stop-and-Wait Protocol for a **Noisy** Channel

- Now we consider the normal situation of a communication channel that makes errors.
  - Frames may be either damaged or lost completely.
  - We assume that if a frame is damaged in transit, the receiver hardware will detect this when it computes the checksum.

- One of the solutions is **to add a timer**. The sender could send a frame, but the receiver would only send an acknowledgement frame if the data were correctly received. If a damaged frame arrived at the receiver, it would be discarded. After a while the sender would time out and send the frame again. This process would be repeated until the frame finally arrived intact.

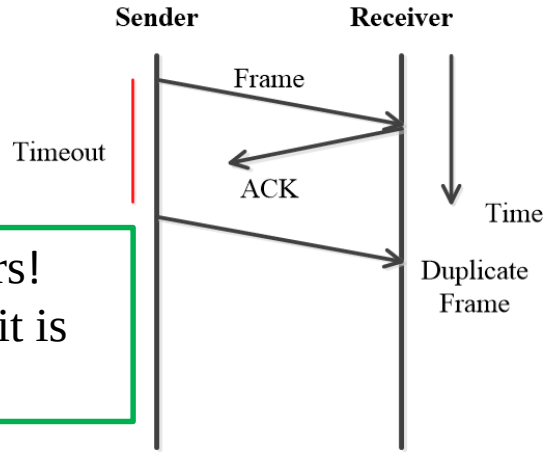
A Simplex Stop-and-Wait Protocol for a **Noisy** Channel

- But this scheme has a fatal flaw in it though. — **duplicate**

1. The network layer on *A* gives packet 1 to its data link layer. The packet is correctly received at *B* and passed to the network layer on *B*. *B* sends an acknowledgement frame back to *A*.
2. The acknowledgement frame gets lost completely. It just never arrives at all. Life would be a great deal simpler if the channel mangled and lost only data frames and not control frames, but sad to say, the channel is not very discriminating.
3. The data link layer on *A* eventually times out. Not having received an acknowledgement, it (incorrectly) assumes that its data frame was lost or damaged and sends the frame containing packet 1 again.
4. The duplicate frame also arrives intact at the data link layer on *B* and is unwittingly passed to the network layer there. If *A* is sending a file to *B*, part of the file will be duplicated (i.e., the copy of the file made by *B* will be incorrect and the error will not have been detected). In other words, the protocol will fail.

Two non-trivial questions about the stop-and-wait protocols

- 1) How long to set the timeout?
  - To allow enough time for the frame to get to the receiver, for the receiver to process it in the worst case, and for the acknowledgement frame to propagate back to the sender.
  - If the timeout interval is set too short, the sender will transmit unnecessary frames.
  - If the timeout interval is set too long, the link will be idle.
- 2) How to avoid accepting **duplicate frames** as new frames?
  - The receiver has to decide whether it is a new frame or a duplicate frame.



Both the frames and the ACKs should carry sequence numbers!
To distinguish the current frame from the next one, a single bit is enough.

A Simplex Stop-and-Wait Protocol for a **Noisy** Channel

- Sequence number
  - The obvious way to achieve this is to have the sender put **a sequence number** in the header of each frame it sends. Then the receiver can check the sequence number of each arriving frame to see if it is a new frame or a duplicate to be discarded.
  - Since the protocol must be correct and the sequence number field in the header is likely to be small to use the link efficiently
    - *Q*: what is the minimum number of bits needed for the sequence number?

Q: what is the minimum number of bits needed for the sequence number ?

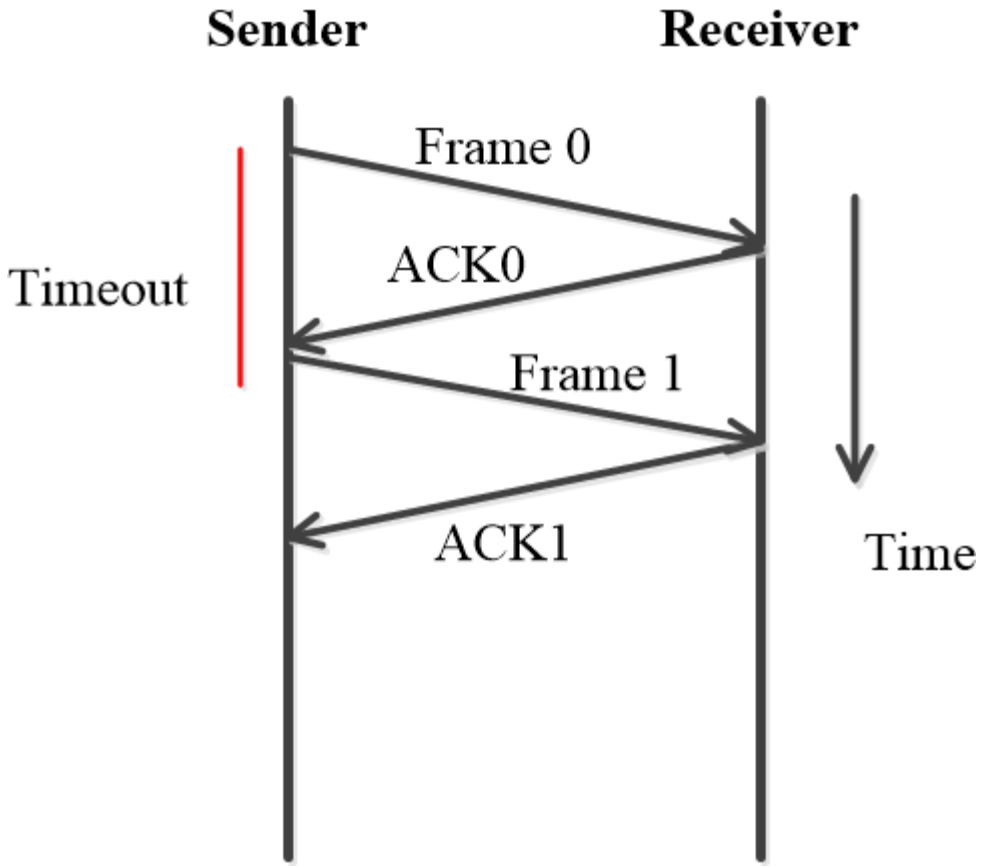
- At the sender, the event that triggers the transmission of frame m+1 is the arrival of an acknowledgement for frame m. This situation implies that m–1 has been correctly received, and furthermore that its acknowledgement has also been correctly received by the sender. Otherwise, the sender would not have begun with m, let alone have been considering m+1.
- As a consequence, the only ambiguity is between a frame and its immediate predecessor or successor, not between the predecessor and successor themselves.
- A **1-bit** sequence number (0 or 1) is therefore sufficient.

A Simplex Stop-and-Wait Protocol for a **Noisy** Channel

- After transmitting a frame and starting the timer, the sender waits for something exciting to happen.
- Only *three* possibilities exist: an acknowledgement frame arrives undamaged, a damaged acknowledgement frame staggers in, or the timer expires.

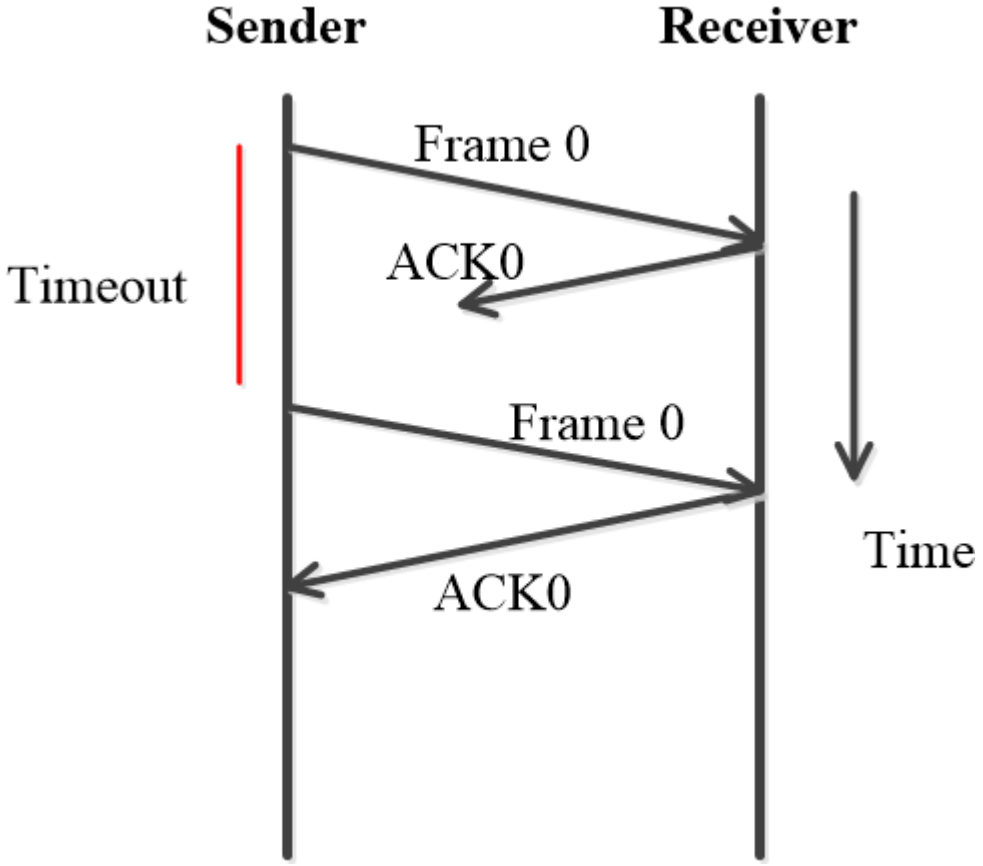
Operations of Stop-and-Wait

- The normal case



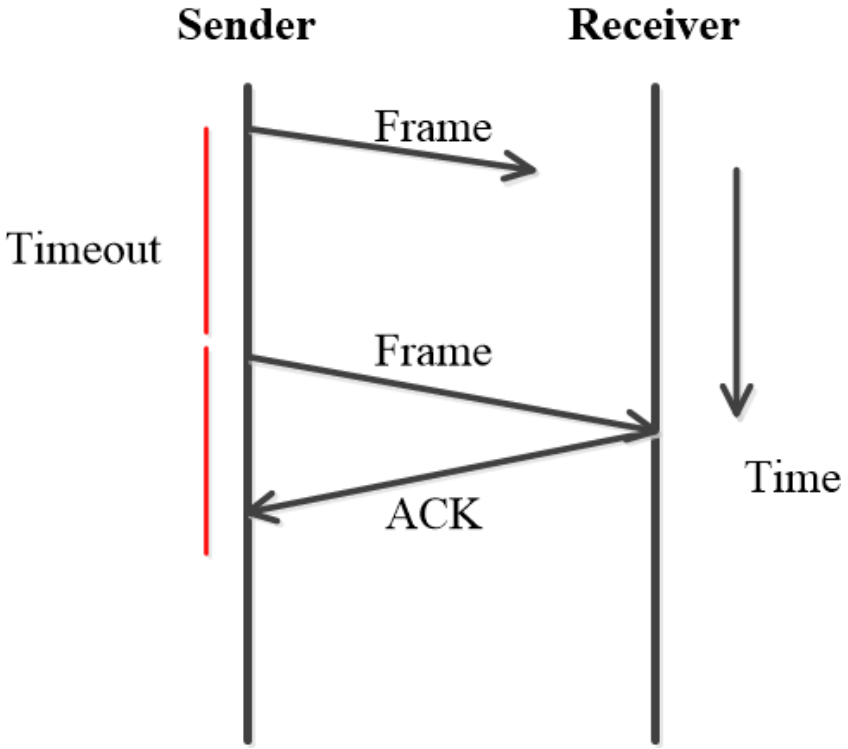
Operations of Stop-and-Wait

- ACK is lost or arrived after the timeout
  - The duplicate frames will not be accepted by the receiver.



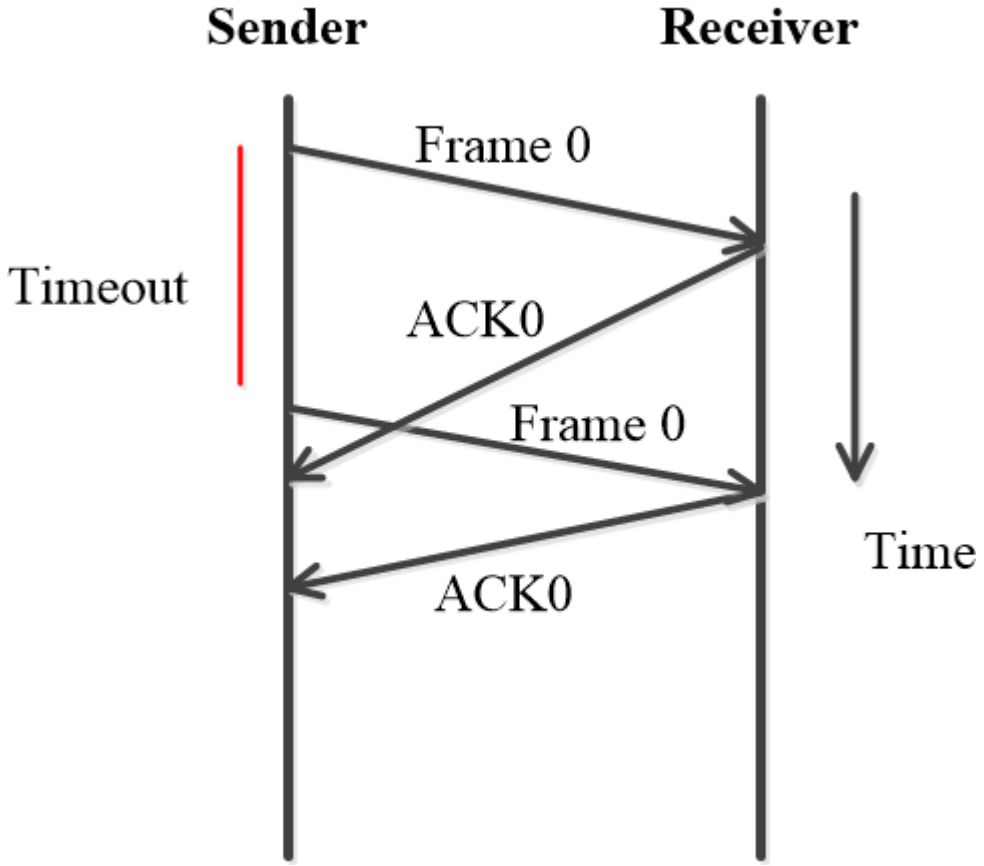
Operations of Stop-and-Wait

- Loss or damaged and retransmission
  - If a frame is lost for multiple times, the sender will repeated again and again.
  - Or if a frame does arrived at the receiver, but it is damaged (does not pass the checksum), then the receiver will not accept it.



Operations of Stop-and-Wait

- ACK arrived after the timeout
  - The duplicate frames will not be accepted by the receiver.



```
#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 */
           }
      }
 }
}
```

1) 如果接受的数据包是期望的那个数据包，则把数据包递交到网络层，同时把frame_expected + 1，（注意这里frame_expected只有两个值：0或1），返回一个ACK，这里ACK应该是1 – frame_expected；
2) 如果接受的数据包不是期望的那个数据包，则返回ACK应该是前一个接受到数据包的序号

```
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.

Outline

- Overview of Data link layer
- Data link layer: **Framing**
- Data link layer: **Error Control**
  - **Error Correction**
  - **Error Detection**
- Elementary data link protocols
- **Sliding window protocols**
- Examples of data link protocols

Outline

- Overview of Data link layer
- Data link layer: **Framing**
- Data link layer: **Error Control**
  - **Error Correction**
  - **Error Detection**
- Elementary data link protocols
- **Sliding window protocols**
- Examples of data link protocols

Sliding Window Protocols

- In the previous protocols, data frames were transmitted in one direction only.
- From now on, the data frames from A to B are intermixed with the acknowledgement frames from A to B.

- **Piggybacking** (驮运)
  - The technique of temporarily delaying outgoing acknowledgements so that they can be hooked onto the next outgoing data frame.
  - In effect, the acknowledgement gets a free ride on the next outgoing data frame.

Q: How long should the data link layer wait for a packet onto which to piggyback the acknowledgement?

- Because the data link layer **cannot** foretell the future, it must resort to some ad hoc scheme, such as waiting for a fixed time interval.
  - If a new packet arrives quickly, the acknowledgement is piggyback onto it. Otherwise, if no new packet has arrived by the end of the sender's timeout period, the data link layer just sends a separate acknowledgement frame.

- Sliding window bidirectional protocol
  - A One-Bit Sliding Window Protocol (stop-and-wait)
  - A Protocol Using Go Back N
  - A Protocol Using Selective Repeat
  - The three differ among themselves in terms of *efficiency*, *complexity* and *buffer requirements*.

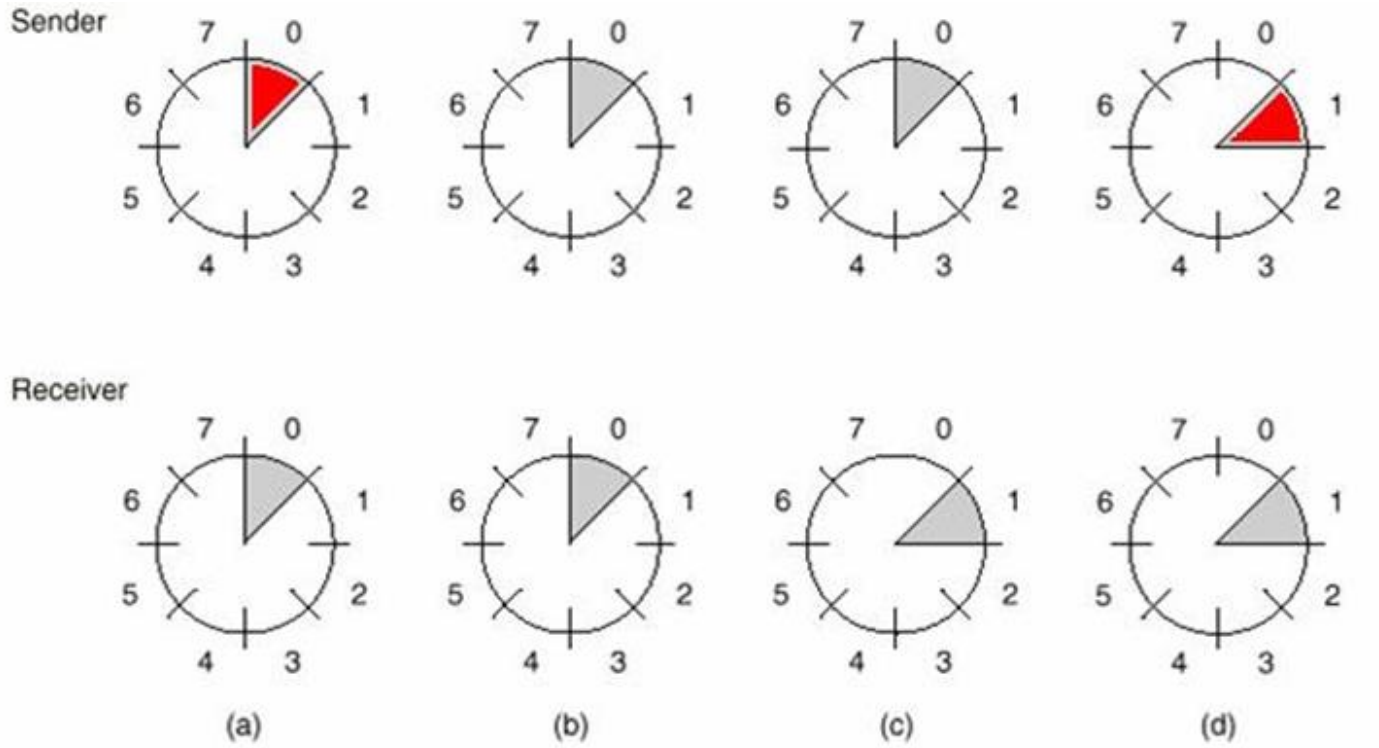
The Essence of All Sliding Window Protocols (I)

- At any instant of time, the sender maintains a set of sequence numbers corresponding to frames it is permitted to send. These frames are said to fall within **the sending window**.
  - The sequence numbers within the sender's window represents frames that 1) have been sent but are yet **not** acknowledged or 2) can be sent. Since frames currently within the sender's window may ultimately be lost or damaged in transit, the sender must keep all of these frames in its memory for possible retransmission.
  - If the maximum window size is *n*, the sender needs *n* buffers to hold the unacknowledged frames. If the window ever grows to its maximum size, the sending data link layer must forcibly *shut off* the network layer until another buffer becomes free. — **Flow Control**

The Essence of All Sliding Window Protocols (II)

- Similarly, the receiver also maintains **a receiving window** corresponding to the set of frames it is permitted to accept.
  - The receiving data link layer's window corresponds to the frames it may accept. Any frame falling within the window is put in the receiver's buffer. Any frame falling outside the window is discarded.
  - When a frame whose sequence number is equal to the lower edge of the window is received, it is passed to the network layer and the window is rotated by one.
  - Note that a window size of 1 means that the data link layer only accepts frames in order, but for larger windows this is not so. (How to keep the correct order?)
- The sender's window and the receiver's window need **not** have the same lower and upper limits or even have the same size.
  - In some protocols they are fixed in size, but in others they can grow or shrink over the course of time as frames are sent and received.

Sliding Window Protocols



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.

One-Bit Sliding Window Protocol

No separated “sender” or “receiver” function here

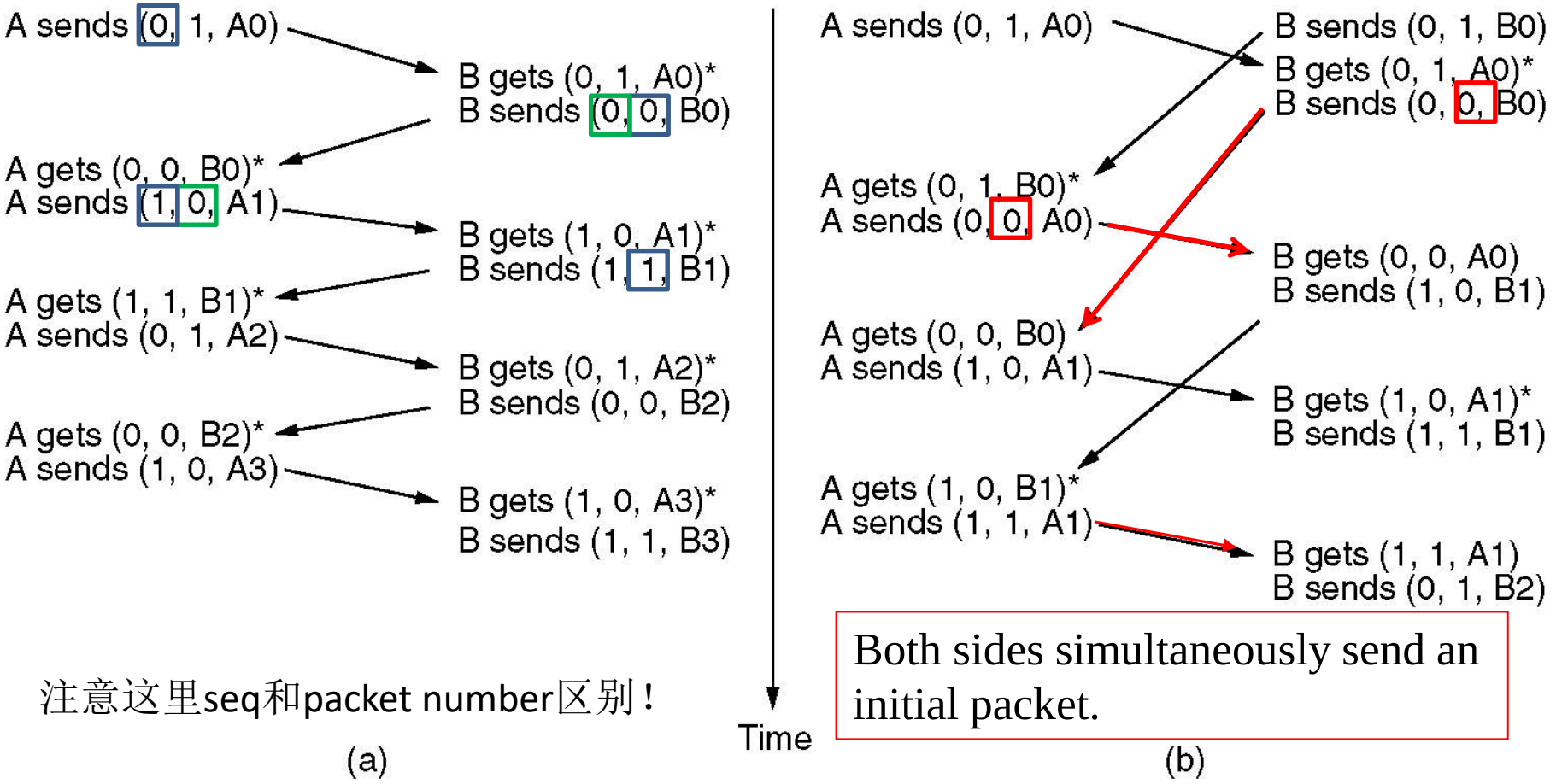
```
#define MAX_SEQ 1                                   /* must be 1 for protocol 4 */
typedef enum {frame_arrival, cksum_err, timeout} event_type;
#include "protocol.h"

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 */
```

Continued →

One-Bit Sliding Window Protocol

```
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 */
}
}
```

One-Bit Sliding Window Protocol



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. (Here the red arrows denotes the retransmissions because of timeout)

An Example

- Example: consider a 50-kbps satellite channel with a 500 msec round-trip **propagation delay**. Let us imagine trying to use protocol 4 to send 1000-bit frames via the satellite.
- At $t = 0$ the sender starts sending the first frame, at $t = 1000$ bit / (50×10^3) bps = 0.02sec = 20 msec (**Transmission delay**) the frame has been completely sent. Not until $t = 500/2 + 20 = 270$ msec has the frame fully arrived at the receiver, and not until $t = 520$ msec has the acknowledgement arrived back at the sender, under the best circumstances.
- But this means that the sender was blocked 500/520 or 96% of the time. In other words, only 4% of the available bandwidth was used.
- The problem described here can be viewed as a consequence of the rule requiring a sender to wait for an acknowledgement before sending another frame.
  - The protocol 4 (One-bit Sliding Window) is disastrous in terms of efficiency under the situation of **a long transit time**, **high bandwidth**, and **short frame length**.

An Example (cont.)

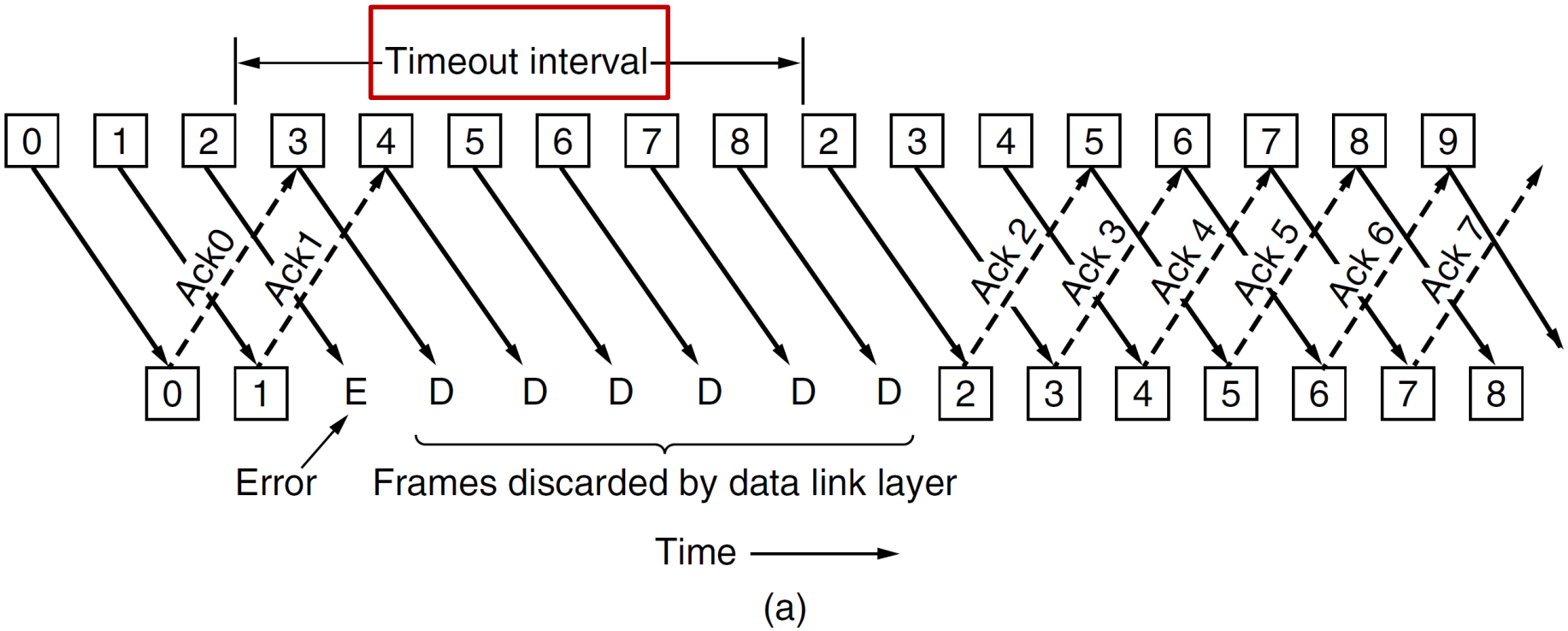
- Basically, the solution lies in allowing the sender to transmit up to *w* frames before blocking, instead of 1.
- How to find an appropriate value for *w*?
  - 1) This capacity is determined by the bandwidth in bits/sec multiplied by the one-way transit (propagation) time, or **the bandwidth-delay product** of the link. $50 \times 10^3 \times 250 \times 10^{-3} = 12.5 \times 10^3$ bits
  - 2) We can divide this quantity by the number of bits in a frame to express it as a number of frames. — $BD = 12.5 \times 10^3$ bits / 1000 bits/frame = 12.5 frames
  - 3) *w* should be set to $2BD + 1$. (w = 26 frames)

Twice the bandwidth-delay is the number of frames that can be outstanding if the sender continuously sends frames when the round-trip time to receive an acknowledgement is considered. The "+1" is because an acknowledgement frame will not be sent until after a complete frame is received.

A Protocol Using Go Back N

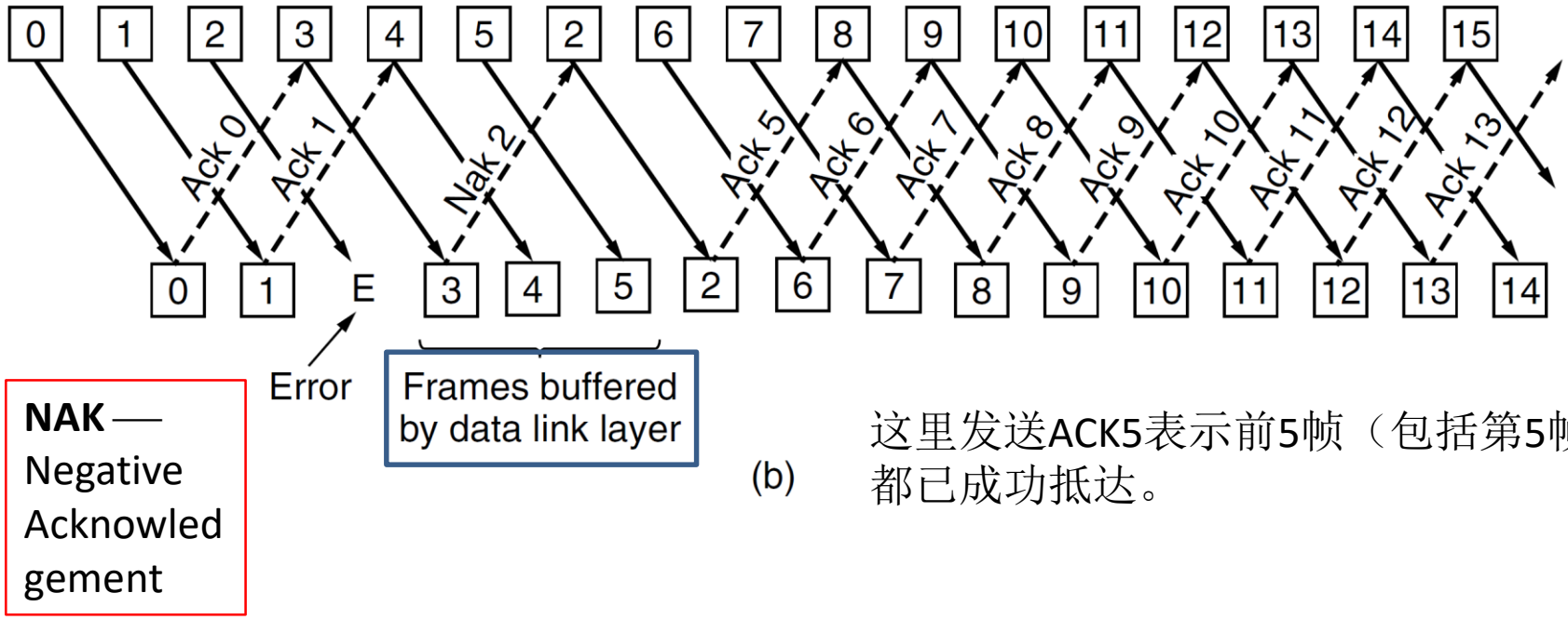
- This technique of keeping multiple frames in flight is an example pipelining. Pipelining frames over an unreliable communication channel raises some serious problem.
  - What happens if a frame in the middle of a long stream is damaged or lost?
  - What should the receiver do with all the correct frames following a damaged frame?
- Remember that the receiving data link layer is obligated to hand packets to the network layer in sequence.

Fig. 3-18 (a) Receiver's window size is 1. — **Go Back N**



The receiver simply discard all subsequent frames, sending no acknowledgements for the discarded frames.

Fig. 3-18 (b) Receiver's window size is large. — **Selective Repeat**



这里发送ACK5表示前5帧（包括第5帧）都已成功抵达。

A bad frame is received and discarded, but any good frames received after it are accepted and buffered. When the sender times out, only the oldest unacknowledged frame is retransmitted.

A Protocol Using Go Back N

- Assumptions:
  - Data in both directions: **piggybacking** of ACK
    - No separate ACK packets
    - Accumulative ACK
  - Noisy channel
  - Limited stream of data from network layer
    - a “network_layer_ready” event

- Protocol: Tanenbaum 5th edition: Fig. 3.19

Sliding Window Protocol Using Go Back N

```
/* 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 */
}
```

Continued →

Sliding Window Protocol Using Go Back N

```
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 */
```

Continued →

Sliding Window Protocol Using Go Back N

```
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 */
         }
```

Continued →

Sliding Window Protocol Using Go Back N

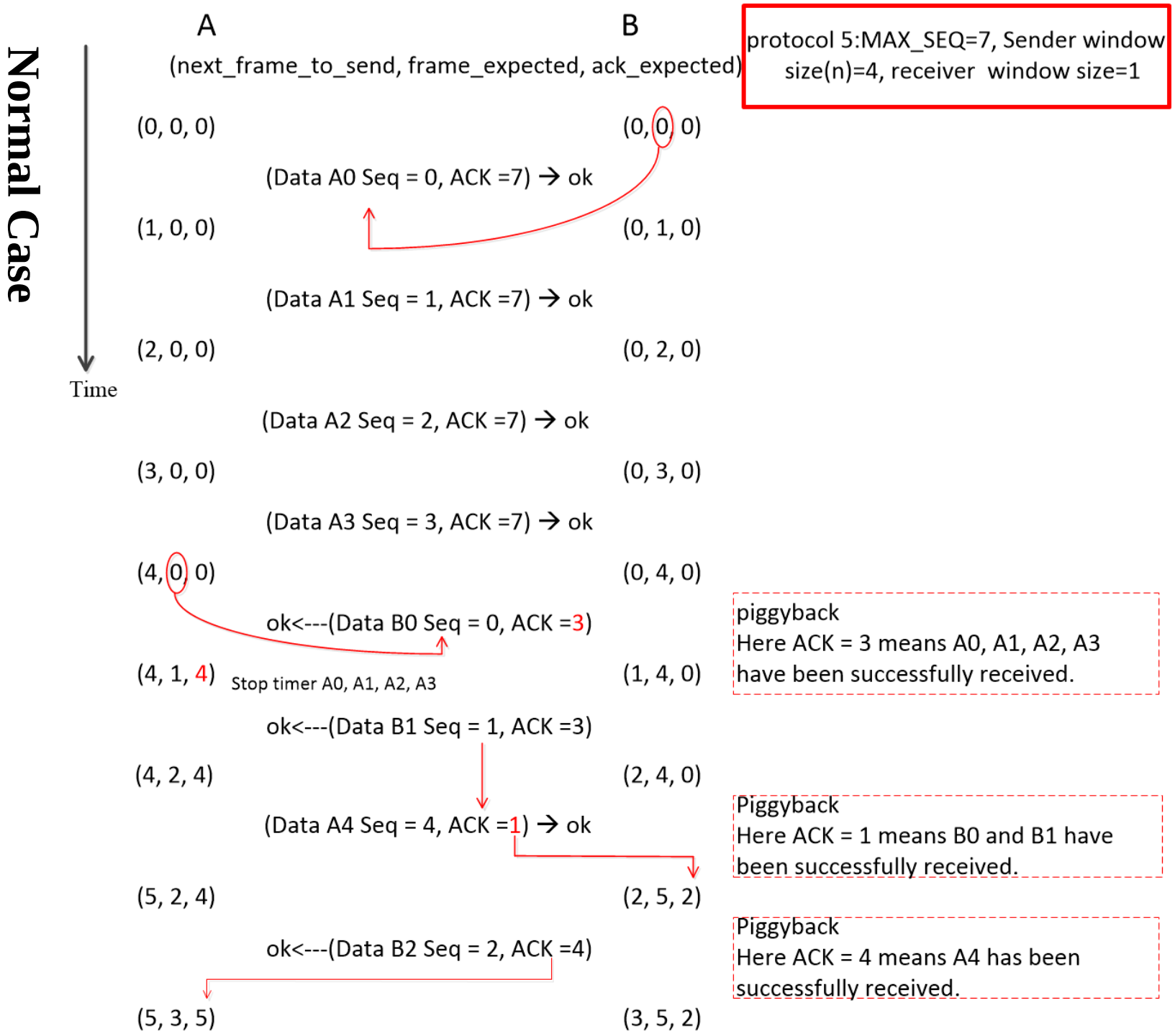
```
        /* Ack n implies n – 1, n – 2, etc.  Check for this. */
        while (between(ack_expected, r.ack, next_frame_to_send)) {
                /* Handle piggybacked ack. */
                nbuffered = nbuffered – 1;          /* one frame fewer buffered */
                stop_timer(ack_expected);           /* 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 */
        }

} // end of switch(event)
```

Please note: in the Go Back N protocol, there is **NO** separate ACK, ACK is **piggybacked** in a data frame.

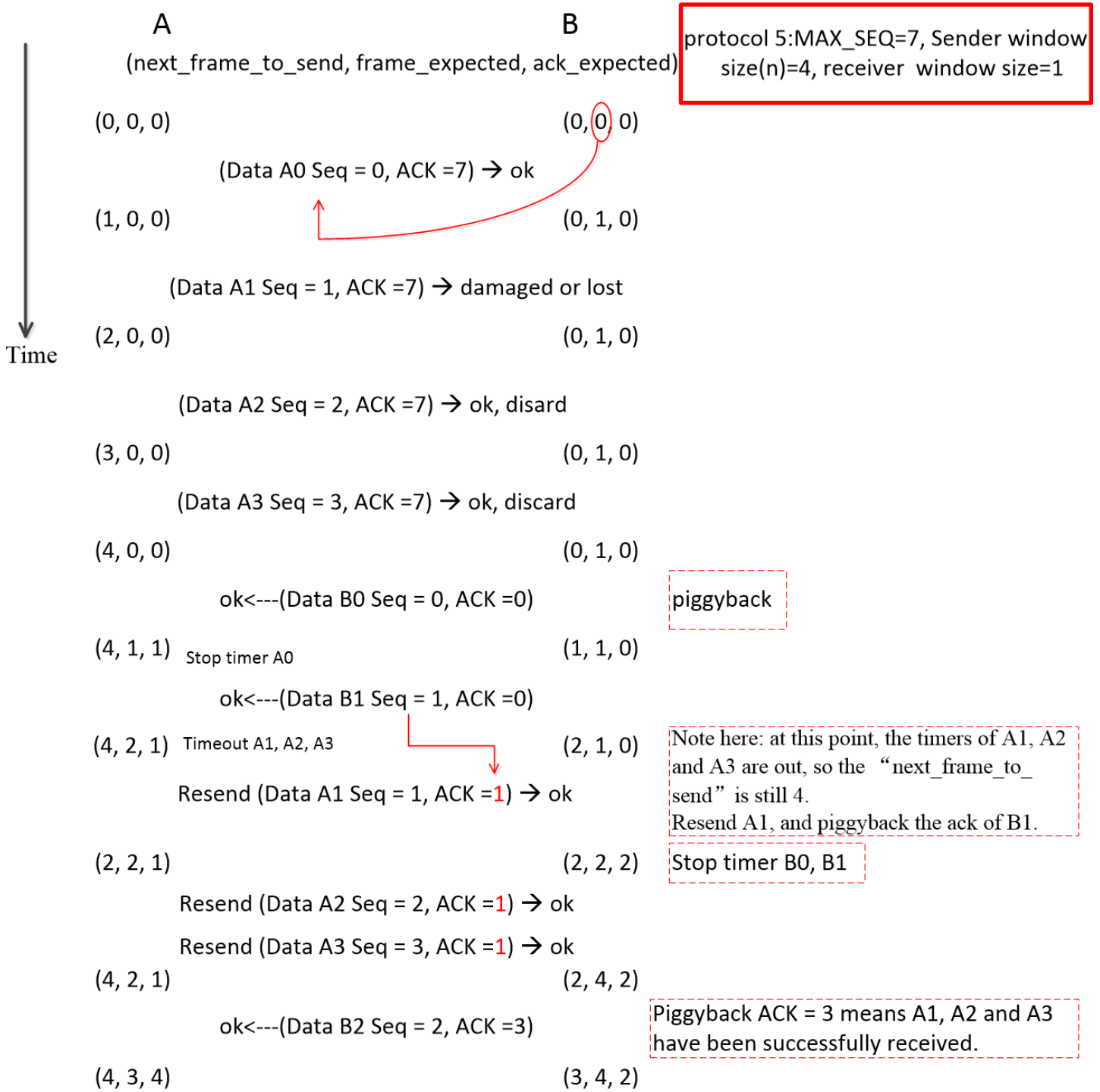
Sliding Window Protocol Using Go Back N

- The sender window of size *n*.
- The receive window of size 1.

- The update rule for next_frame_to_send, frame_expected, ack_expected:
  - ***next_frame_to_send*** is increased every time a new or old frame is sent, or go back to ack_expected when timeout occurs.
  - ***frame_expected*** is increased every time an expected frame is received.
  - ***ack_expected*** is increased by *n* when an incoming frame acknowledged *n* outstanding frames which are waiting for acknowledgement.


Normal Case
A
B
protocol 5:MAX_SEQ=7, Sender window size(n)=4, receiver window size=1
(next_frame_to_send, frame_expected, ack_expected)
(0, 0, 0)
(0, 0, 0)
(Data A0 Seq = 0, ACK =7) → ok
(1, 0, 0)
(0, 1, 0)
(Data A1 Seq = 1, ACK =7) → ok
(2, 0, 0)
(0, 2, 0)
Time
(Data A2 Seq = 2, ACK =7) → ok
(3, 0, 0)
(0, 3, 0)
(Data A3 Seq = 3, ACK =7) → ok
(4, 0, 0)
(0, 4, 0)
ok<---(Data B0 Seq = 0, ACK =3)
piggyback
Here ACK = 3 means A0, A1, A2, A3 have been successfully received.
(4, 1, 4)
Stop timer A0, A1, A2, A3
(1, 4, 0)
ok<---(Data B1 Seq = 1, ACK =3)
(4, 2, 4)
(2, 4, 0)
(Data A4 Seq = 4, ACK =1) → ok
Piggyback
Here ACK = 1 means B0 and B1 have been successfully received.
(5, 2, 4)
(2, 5, 2)
ok<---(Data B2 Seq = 2, ACK =4)
Piggyback
Here ACK = 4 means A4 has been successfully received.
(5, 3, 5)
(3, 5, 2)

Damaged or Lost Case



Sliding Window Protocol Using Go Back N

- MAX_SEQ+1 distinct sequence numbers(0, 1, 2, ... MAX_SEQ), no more than MAX_SEQ unacknowledged frames, the sender window<=MAX_SEQ.Why?

- Consider the following scenario with MAX_SEQ=7 (a sender window of size 8):
- 1) The sender sends frames 0 through 7.
- 2) A piggybacked acknowledgement for frame 7 eventually comes back to the sender.
- 3) The sender sends another 8 frames, again with sequence numbers 0 through 7.
  - All eight frames belonging to the second batch get lost
  - All eight frames belonging to the second batch arrive successfully.
- 4) Another piggybacked acknowledgement for frame 7 come in, **ambiguity** will arise to sender:
  - All eight frames belonging to the second batch get lost,ack=7
  - All eight frames belonging to the second batch arrive successfully, ack=7

Sliding Window Protocol Using Go Back N

- When the sender window of size = 7
  - The sender sends frames 0 through 6
  - A piggybacked acknowledgement for frame 6 eventually comes back to the sender
  - The sender sends another 7 frames:7,0,1,2,3,4,5,
    - Frame 7,0,1,2,3,4,5 belonging to the second batch get lost
    - Frame 7,0,1,2,3,4,5 belonging to the second batch arrive
  - Another piggybacked acknowledgement comes in, no ambiguity
    - Frame 7,0,1,2,3,4,5 belonging to the second batch get lost, ack=6
    - Frame 7,0,1,2,3,4,5 belonging to the second batch arrive, ack=5

Sliding Window Protocol Using Go Back N

- Because protocol 5 has multiple outstanding frames, it *logically* needs multiple timers, one per outstanding frames. Each frame times out *independently* of all the other ones.

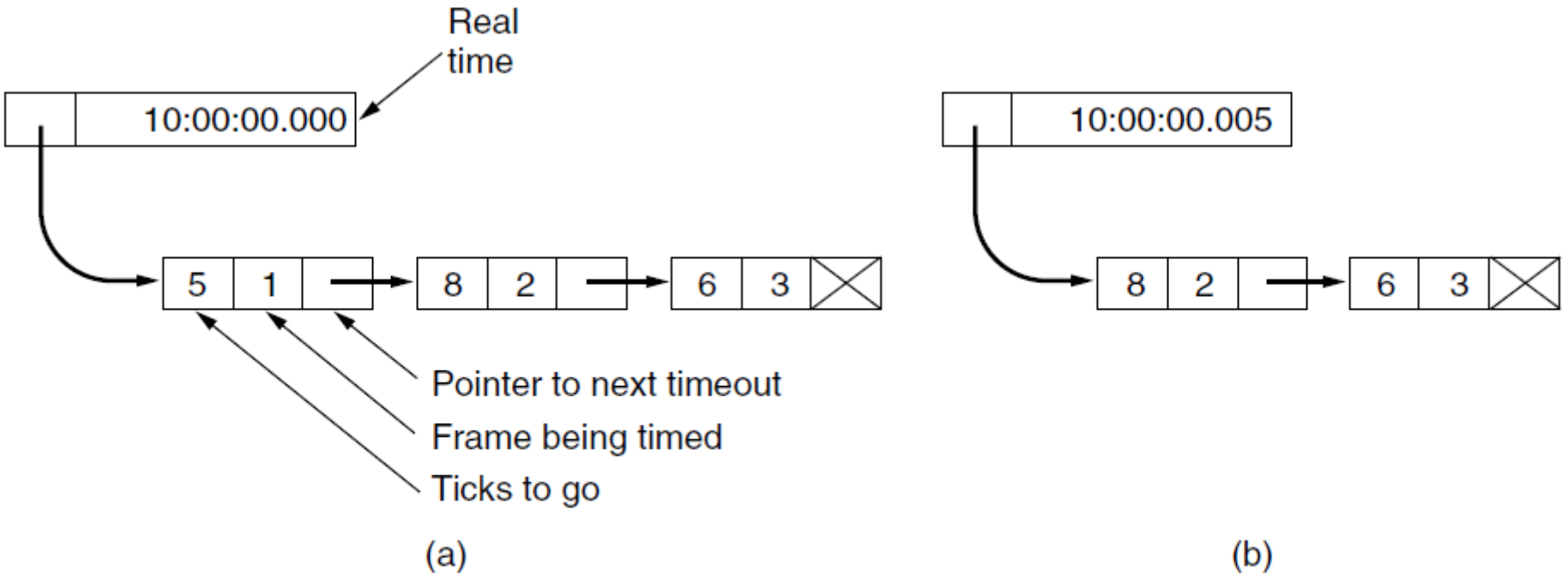


Figure 3-20. Simulation of multiple timers in software. (a) The queued timeouts. (b) The situation after the first timeout has expired.

A Sliding Window Protocol Using Selective Repeat

- The **go-back-N** protocol works well *if errors are rare*, but if the line is poor it wastes a lot of bandwidth on retransmitted frames.
  - The selective repeat protocol is to allow the receiver to accept and buffer the frames following a damaged or lost one.
    - The receiver accepts frames out of order but passes packets to the network layer in order

- Assumptions:
  - Data in both directions: piggybacking of ACK
    - **Separate** ACK packets (requires ACK timer)
  - Noisy channel
  - Limited stream of data from network layer
  - NACK packets: receiver detected problem

- Protocol: Tanenbaum 5th Edition Fig. 3.21

A Sliding Window Protocol Using Selective Repeat

- In this protocol, both sender and receiver maintain a window of outstanding and acceptable sequence numbers, respectively.
- 1) The sender's window size starts out at 0 and grows to some predefined maximum.
- 2) The receiver's window, in contrast, is always **fixed** in size and equal to the predetermined maximum.
- 3) The receiver has a buffer reserved for each sequence number within its fixed window. Associated with each buffer is a bit (**arrived**) telling whether the buffer is full or empty. Whenever a frame arrives, its sequence number is checked by the function "*between*" to see if it falls within the window. If so and if it has not already been received, it is accepted and stored.
  - **Nonsequential receive** introduces further constraints on frame sequence number compared to protocols in which frames are only accepted in order!

Problems with the Go-back-N (Protocol 5)

- In Protocol 5 (go-back-N), the number of timers needed is equal to the number of buffers in the sender, not to the size of the sequence space; and no separate acknowledgement (just piggyback).
  - If the reverse traffic is light, the acknowledgements may be held up for a long period of time, which can cause problems.
  - In the extreme, if there is a lot of traffic in one direction and no traffic in the other direction, the protocol will block when the sender window reaches its maximum.
- To relax this assumption, an auxiliary timer is started by *start_ack_timer* after an in-sequence data frame arrives.
  - If no reverse traffic has presented itself before this timer expires, a **separate** acknowledgement frame is sent.
  - An interrupt due to the auxiliary timer is called an **ack_timeout** event.

Problems with the Go-back-N (Protocol 5)

- Protocol 6 (selective repeat) uses a more efficient strategy than protocol 5 for dealing with errors.
  - When a damaged frame arrives or a frame other than the expected one arrives (potential lost frame), the receiver sends *a negative acknowledgement* (**NAK**) frame back to the sender. Such a frame is a request for retransmission of the frame specified in the NAK.
  - To avoid making multiple requests for retransmission of the same lost frame, the receiver should keep track of whether a NAK has already been sent for a given frame.
    - The variable *no_ack* in protocol 6 is **true** if **no NAK** has been sent for a given frame.
    - If the NAK gets mangled or lost, no real harm is done, since the sender will eventually time out and retransmit the missing frame.

A Sliding Window Protocol Using Selective Repeat

- Selective repeat protocol
  - Maximum size of sender window?
    - Sequence numbers
      - 0 .. MAXSEQ
    - Size? MAXSEQ ?

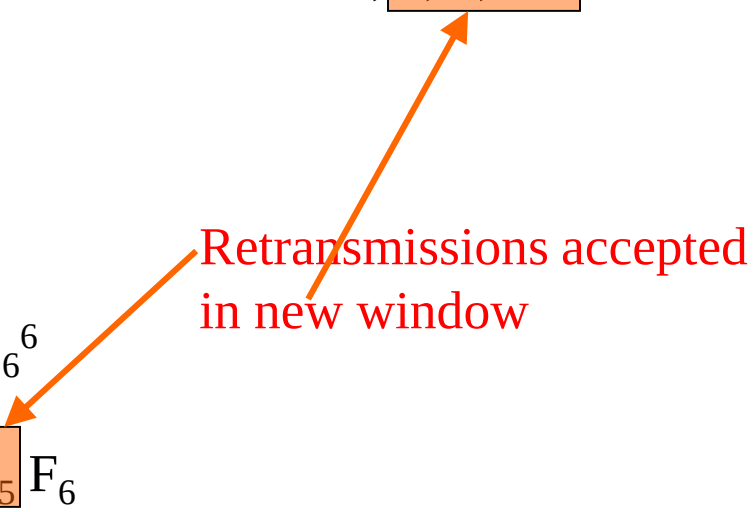
A Sliding Window Protocol Using Selective Repeat

- Selective repeat protocol: Maximum size = MAXSEQ?
  - Sender sends $F_0 \ldots F_6$ or $F_0^0 \ldots F_6^6$
  - Receiver sends Ack_6

    receive window: 7, 0, 1,…5

    and expects $F_7 \ldots$ or $F_7^7 \ldots$.
  - Ack6 lost
  - Sender retransmits $F_0^0 \ldots F_6^6$

    or $F_0 \ldots F_5 F_6$



The essence of the problem is that after the receiver advanced its window, the new range of valid sequence numbers **overlapped** the old one.

A Sliding Window Protocol Using Selective Repeat

- Selective repeat protocol: Max. size = (MAXSEQ+1)/2
  - Sender sends F_0 … F_3 or F_0^0 … F_3^3
  - Receiver sends Ack_3

    receive window: 4, 5, 6, 7

    and expects F_4 … or F_4^4 ….
  - Ack_3 lost
  - Sender retransmits F_0^0 … F_3^3

    or F_0 …F_2 F_3

No overlap with new window

A Sliding Window Protocol Using Selective Repeat

```
/* 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 */
}
```

Continued →

A Sliding Window Protocol Using Selective Repeat (2)

```
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;
                                          /*initialization of the arrival bit*/
```

Continued →

A Sliding Window Protocol Using Selective Repeat (3)

```
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)  /* no_nak == true */
                 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 */
                   }
              }
         }
```

Continued →

A Sliding Window Protocol Using Selective Repeat (4)

```
        if((r.kind==nak) && between(ack_expected,(r.ack+1)%(MAX_SEQ+1),next_frame_to_send))
            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 */
  } // end switch(event)
  if (nbuffered < NR_BUFS) enable_network_layer(); else disable_network_layer();
 }    // end while(true)
}
```

Outline

a) Overview of Data link layer

b) Data link layer: **Framing**

c) Data link layer: **Error Control**

- **Error Correction**
- **Error Detection**

d) Elementary data link protocols

e) Sliding window protocols

f) Examples of data link protocols

Outline

a) Overview of Data link layer

b) Data link layer: **Framing**

c) Data link layer: **Error Control**

- **Error Correction**
- **Error Detection**

d) Elementary data link protocols

e) Sliding window protocols

f) Examples of data link protocols

Example Data Link Protocols

- We will examine the data link protocols found on point-to-point lines in the internet in two common situations:
  - Packets are sent over SONET optical fiber links in wide-area networks
    - To connect routers in the different locations of an ISP's network.
  - ADSL (Asymmetric Digital Subscriber Loop)
    - ADSL links running on the local loop of the telephone network at the edge of the internet.

Point-to-Point Protocol

- A standard protocol called PPP (Point-to-Point Protocol) is used to send packets over these links.
  - PPP works in **the data link layer protocol**.
  - PPP is defined in RFC 1661 and elaborated in RFC 1662.
  - PPP is a framing mechanism that can carry the packets of multiple protocols over many types of physical layers.
  - SONET and ADSL links both apply PPP but in different ways.

- PPP provides three main features:
  - A **framing** method that unambiguously delineates the end of one frame and the start of the next one. The frame format also handles **error detection**.
  - A **link control protocol** for bringing lines up, testing them, negotiating options, and bringing them down again gracefully when they are no longer needed. This protocol is called **LCP** (Link Control Protocol).
  - A way to negotiate network-layer options in a way that is **independent of the network layer protocol** to be used. The method chosen is to have a different **NCP** (**Network Control Protocol**) for each network layer supported.

Packet over SONET (I)

- SONET is *the physical layer protocol* that is most commonly used over the wide-area optical fiber links that make up the backbone of communication networks, including the telephone system.
  - It provides a bitstream that runs at a well-defined rate.
  - This bitstream is organized as *fixed-size byte* payloads that *recur every 125μsec*, whether or not there is user data to send.

Packet over SONET (II)

- PPP runs on **IP router** to provide the transmission mechanism.

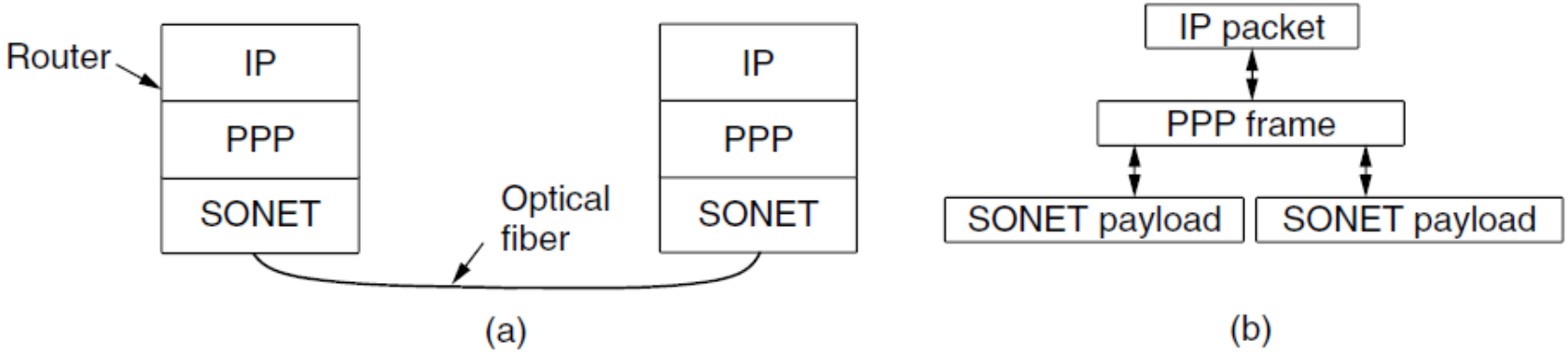


Figure 3-23. Packet over SONET. (a) A protocol stack. (b) Frame relationships.

- The PPP **frame format**
  - PPP uses **byte stuffing** and all frames are an integral number of bytes.

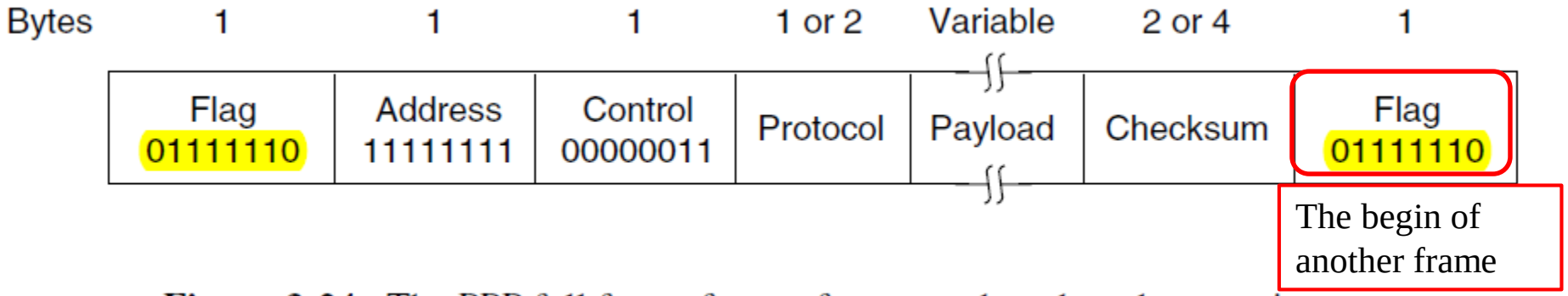


Figure 3-24. The PPP full frame format for unnumbered mode operation.

The PPP Frame Format

- All PPP frames begin with the standard **HDLC** flag byte of **0x7E** (01111110). The flag byte is stuffed if it occurs within the payload field using the escape byte **0x7D**.
  - ESC FLAG → 0x7D 0x7E → 0x7D (0x7E XOR 0x20) → 0x7D 0x5D (~ bit stuffing)
  - This means the start and end of frames can be searched for simply by scanning for the byte 0x7E since it will not occur elsewhere.
  - The destuffing rule when receiving a frame is to look for 0x7D, remove it, and XOR the following byte with 0x20 (~ destuffing).
- The **Address** field — This field is always set to the binary value 11111111 to indicate that all stations are to accept the frame.
- The **Control** field — The default value is 00000011. This value indicates *an unnumbered frame*.
  - Used in the internet to provide connectionless unacknowledged services

The PPP Frame Format (II)

- The **Protocol** field — To tell what kind of packet is in the payload field. (its default size is 2 bytes)
  - Codes starting with a 0 bit are defined for IPv4, IPv6, and other network layer protocols
  - Codes starting with a 1 bit are used for PPP configuration protocols.
- The **Payload** field is variable length, up to some negotiated maximum. (default length of 1500 bytes)
  - The PPP payload is *scrambled* before it is inserted into the SONET payload.
    - Scrambling: XORs the payload with a long pseudorandom sequence before it is transmitted.
    - The SONET bitstream needs frequent bit transition for synchronization.
      - These transitions come naturally with the variation in voice signals, but in data communication the user chooses the information that is sent and might send a packet with a long run of 0s.
- The **Checksum** field which normally is 2 bytes

PPP Link Up & Down (I)

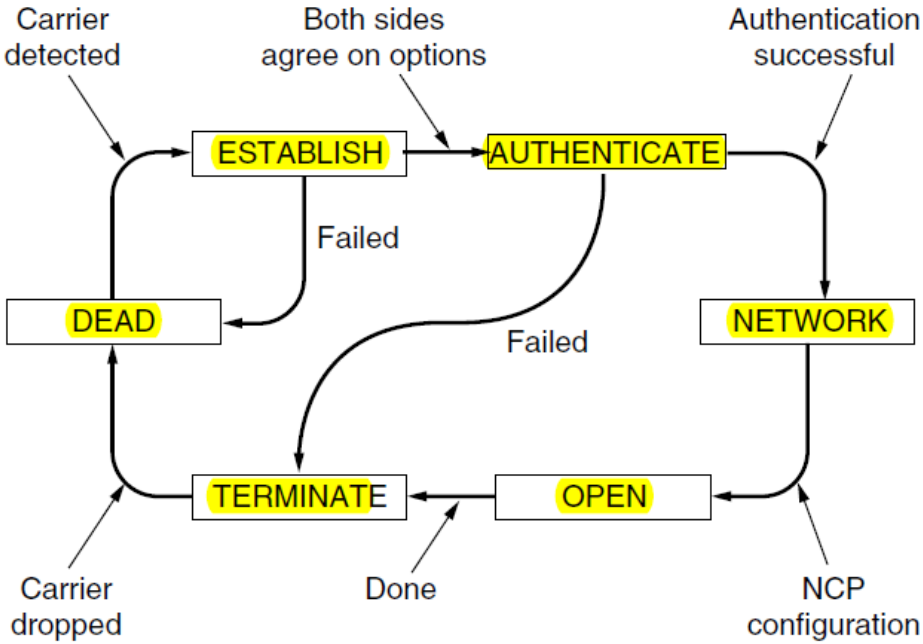


Figure 3-25. State diagram for bringing a PPP link up and down.

- The link starts in the **DEAD** state, which means that there is no connection at the physical layer.
- When a physical connection is established, the link moves to **ESTABLISH**. At this point, the PPP peers exchange a series of **LCP** packets.
- If LCP option negotiation is successful, the link reaches the **AUTHENTICATE**.

PPP Link Up & Down (II)

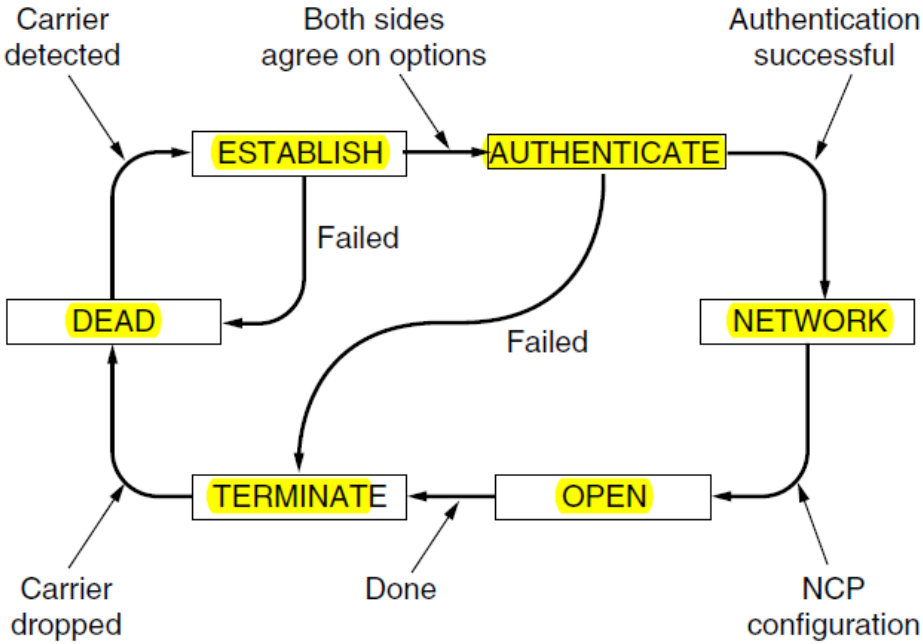


Figure 3-25. State diagram for bringing a PPP link up and down.

- If authentication is successful, the **NETWORK** state is entered and a series of **NCP packets** are sent to configure the network layer. It is difficult to generalize about the NCP protocols because each one is specific to some network layer protocol.
- Once **OPEN** is reached, data transport can take place. It is in this state that **IP packets** are carried in PPP frames across the SONET line.
- When data transport is finished, the link move to the **TERMINATE** state, and from there it moves back to the **DEAD** state when the physical layer connection is dropped.

ADSL (Asymmetric Digital Subscriber Loop)

- The ADSL physical layer based on OFDM digital modulation
  - At the DSLAM, the IP packets are extracted and enter an ISP network so that they may reach any destination on the Internet.
  - PPP works in the same way to establish and configure the link and carry IP packets.
  - In between ADSL and PPP are ATM and AAL5.

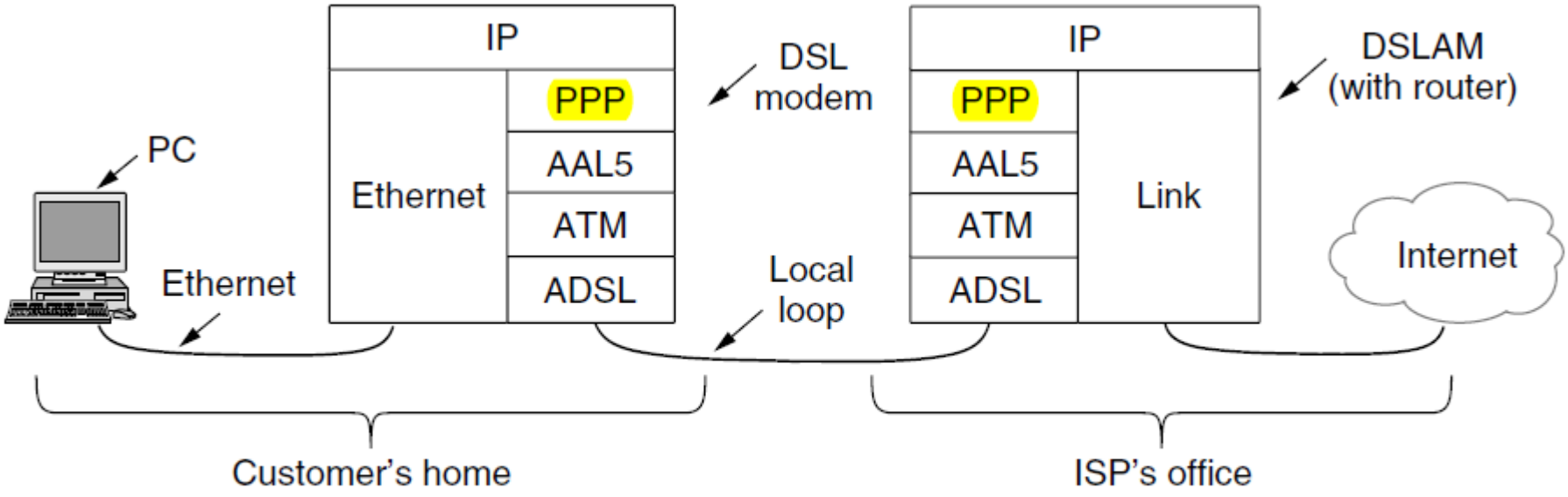


Figure 3-26. ADSL protocol stacks.

ADSL (Asymmetric Digital Subscriber Loop)

- ATM (Asynchronous Transfer Mode) is a link layer that is based on the transmission of fixed-length **cells** of information.
  - These cells may contain voice, data, TV signals, …
  - The "Asynchronous" in its name means that the cells do not always need to be sent in the way that bits are continuously sent over synchronous lines, as in SONET.
  - ATM is a **connection-oriented** technology. Each cell carriers a **virtual circuit identifier** in its header and devices use this identifier to forward cells along the paths of established connections.
  - The cells are each 53 bytes long, consisting of a 48-byte payload plus a 5-byte header.
  - To send data over an ATM network, it needs to be mapped into a sequence of cells. This mapping is done with an ATM adaptation layer in a process called segmentation and reassembly.
    - Several adaptation layers have been defined for different services.
    - The main one for packet data is **AAL5** (**ATM Adaptation Layer 5**)

ADSL (Asymmetric Digital Subscriber Loop)

- An AAL5 frame
  - No addresses are needed on the frame as **the virtual circuit identifier** carried in each cell will get it to the right destination.
  - It has a trailer that gives length and has a 4-byte CRC for error detection. The CRC is the same one used for PPP and IEEE 802 LANs.
  - The payload of the AAL5 frame has padding to be a multiple of 48 bytes so that the frame can be evenly divided into cells.
  - The protocol field indicates to the DSLAM at the far end whether the payload is an IP packet or a packet from other protocol such as LCP.

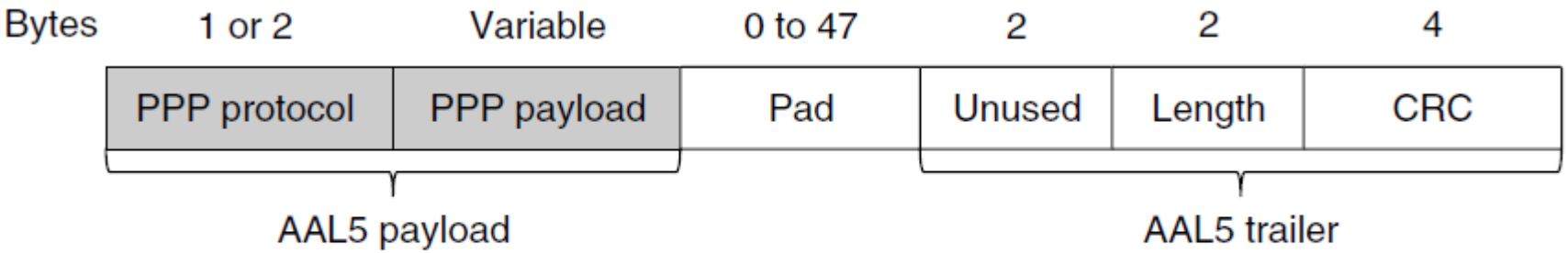


Figure 3-27. AAL5 frame carrying PPP data.

Data Over Cable Service Interface Specification (**DOCSIS**)

- The DOCSIS protocol is generally described as having two components: the physical layer (or the PMD, physical media dependent sublayer), and the MAC layer (in Chapter 4).

- Above the physical layer, DOCSIS must handle a variety of tasks for the network layer:
  - Bandwidth allocation in the upstream and downstream direction (flow control)
  - Framing
  - error correction.

- Once **a cable modem** has been powered on, it establishes a connection to **the CMTS**. It acquires upstream and downstream communication channels to use, as well as encryption keys from the CMTS.
  - The upstream and downstream carriers provide two **shared** channels for all cable modems.

Data Over Cable Service Interface Specification (**DOCSIS**) [5]

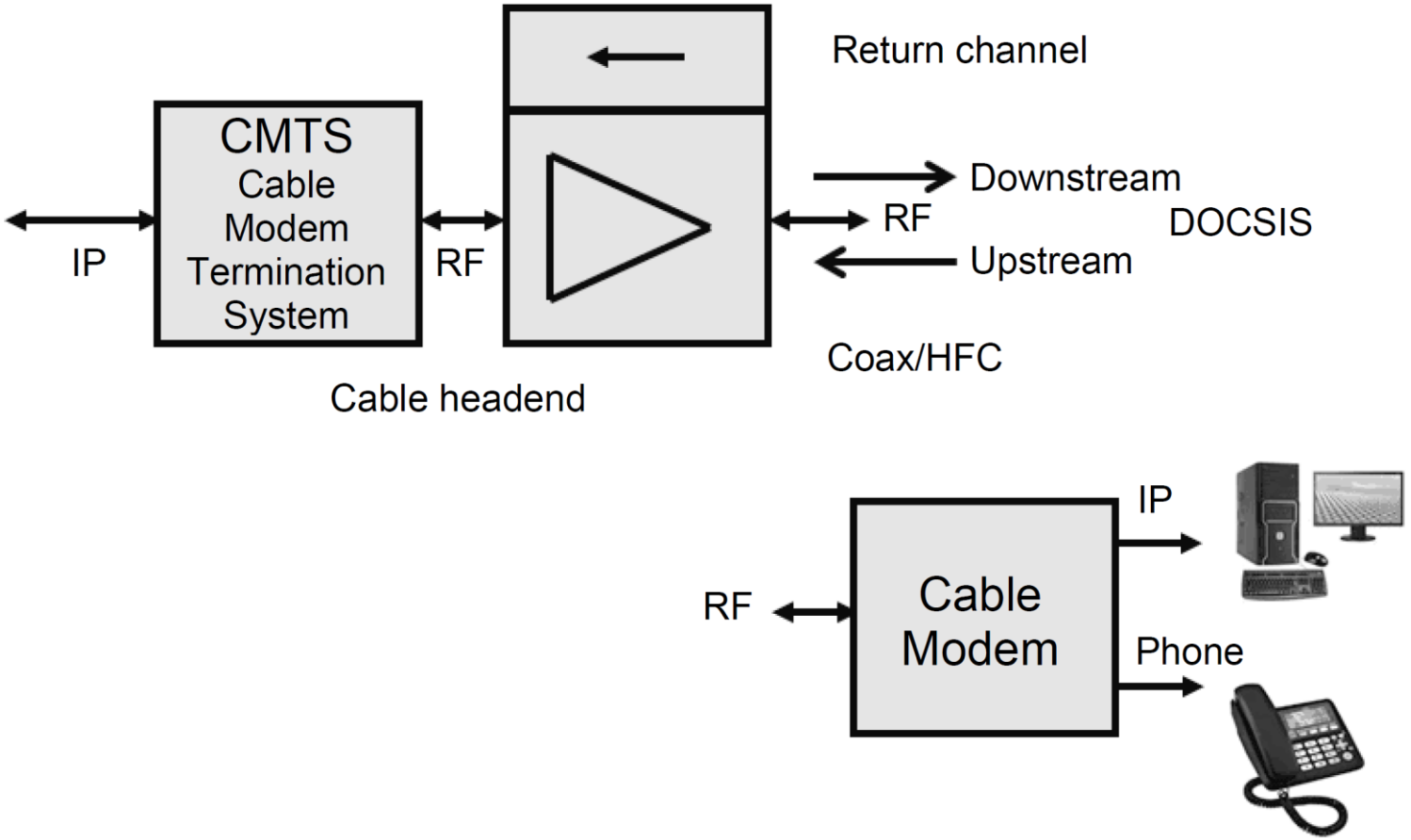


Fig. 33.1. CATV network with cable modem termination system (CMTS) and cable modem (CM)

Data Over Cable Service Interface Specification (**DOCSIS**) [5]

Table 33.2. Technical parameters of the DOCSIS downstream

Downstream parameter	DOCSIS 3.1	DOCSIS 1.0 … 3.0
Modulation type	OFDM 4K and 8K FFT	single carrier (J.83/B, DVB-C)
Frequency range	108 MHz … 1218 MHz (1794 MHz)	45 MHz … 1.002 MHz
Channel bandwidth	up to 192 MHz	6 MHz / 8 MHz
QAM systems	up to 4096 (8k, 16k)	up to 256
Cyclic prefix length (guard interval)	0.9375 µs to 5 µs	--
Pilots	scattered and continuous	--
Error protection	BCH-LDPC	Reed-Solomon
Data rate	8 Gbit/s (10 Gbit/s)	300 Mbit/s (1 Gbit/s)

Data Over Cable Service Interface Specification (**DOCSIS**) [5]

Table 33.3. Technical parameters of the DOCSIS upstream

Upstream parameter	DOCSIS 3.1	DOCSIS 3.0
Modulation type	OFDM 2K and 4K FFT	single carrier TDMA, S-CDMA
Frequency range	5 MHz … 204 MHz	5 MHz … 85 MHz
Channel bandwidth	up to 96 MHz	up to 6.4 MHz
QAM systems	up to 4096	QPSK … 64QAM
Cyclic prefix length (guard interval)	0.9375 µs … 6.25 µs	--
Pilots	complementary and continuous	--
Error protection	LDPC	Reed-Solomon
Data rate	400 Mbit/s (1 Gbit/s ... 2.5 Gbit/s)	100 Mbit/s (300 Mbit/s)

Data Over Cable Service Interface Specification (**DOCSIS**)

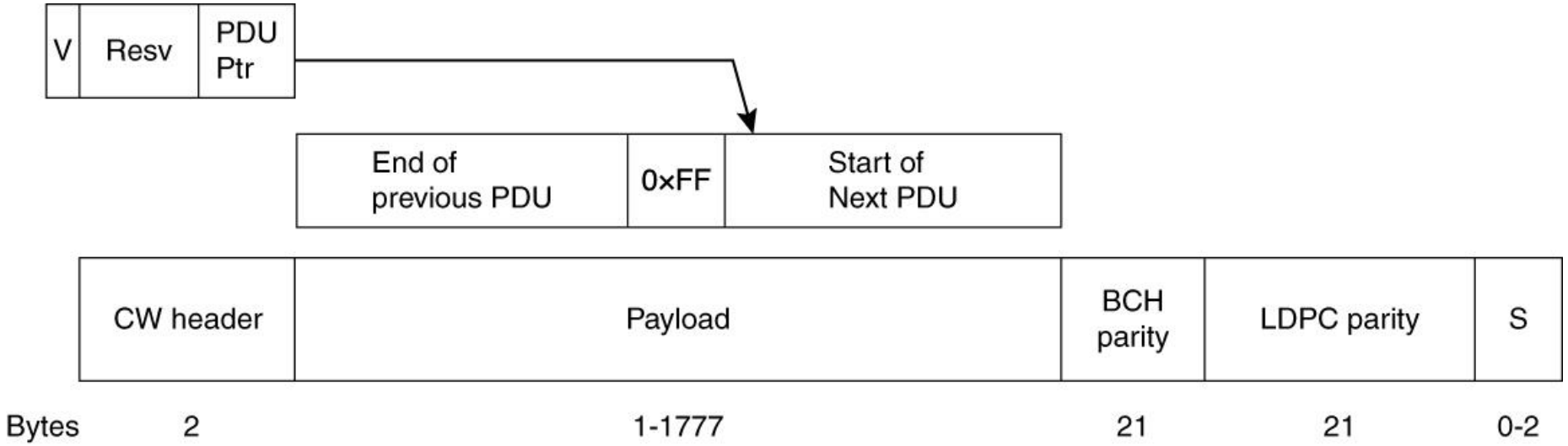
- Prior to DOCSIS 3.1, packets in the downstream direction were divided into **188-byte MPEG frames**, each with a 4-byte header and a 184-byte payload (the so-called MPEG transmission convergence layer).
- In addition to the data itself, the CMTS periodically sends management information to the cable modem, which includes information about ranging, channel assignment, and other tasks related to channel allocation.

Data Over Cable Service Interface Specification (**DOCSIS**)

- The DOCSIS link layer organizes transmission according to **modulation profiles**.
  - A modulation profile is a list of modulation orders (i.e., bit-loading) that correspond to the OFDM subcarriers.
- Based on the service flow identification and QoS parameters, the link layer (in DOCSIS 3.1), now called **the convergence layer**, groups packets that have the same profile into same send buffer.
  - Typically there is one send buffer per profile, each of which is shallow so as to avoid significant latency.
- The codeword builder then maps each DOCSIS frame to the corresponding FEC codewords.
  - BCH and LDPC error protection in the downstream
  - LDPC error protection in the upstream

Data Over Cable Service Interface Specification (**DOCSIS**)

- The CMTS adopts **byte stuffing**: if no DOCSIS frame is available in the downstream direction, the CMTS inserts zero-bit-filled subcarriers into OFDM symbols, or simply stuffs sequences of 1s into codewords.
- Since version 3.0, DOCSIS has supported a technology called **channel bonding**, which allows a single subscriber to use multiple upstream and downstream channels simultaneously.



Main Points (I)

- Three main functions of the data link layer
  - Framing
    - 1) Byte Count
    - 2) Flag bytes with byte stuffing
    - 3) Flag bits with bit stuffing
    - 4) Physical layer coding violations
  - Dealing with errors
    - Error detection: parity check, the 16-bit Internet checksum, CRC
    - Error correction: Hamming codes, convolutional code
    - Hamming distance between two binary sequences vs. the Hamming distance of the complete code
  - Flow control

Main Points (II)

- Sliding window bidirectional protocol
  - A One-Bit Sliding Window Protocol
  - A Protocol Using Go Back N (No separated ACK!)
  - A Protocol Using Selective Repeat
  - **Piggybacking**

- Two fundamentally different types of link-layers channels
  - Broadcast channels (Chapter 4)
  - The point-to-point communication links (Two examples: SONET, ADSL)

References

- [1] A.S. Tanenbaum, and D.J. Wetherall, Computer Networks, 5th Edition, Prentice Hall, 2011.
- [2] G.D. Forney Jr., “The Viterbi algorithm,” Proc. of the IEEE, vol.61, no.3, pp.268-278, 1973.
- [3] https://www.cisco.com/c/zh_cn/tech/wan/point-to-point-protocol-ppp/index.html
- [4] J. F. Kurose and K.W. Ross, Computer Networking — A Top-down Approach, 5th Edition, Pearson Education Inc., 2010.
- [5] W. Fischer, “DOCSIS – Data over Cable Service Interface Specification,” in Book: Digital Video and Audio Broadcasting Technology, Springer, 2020.