Chapter 3

The Data Link Layer

Many protocols/algorithms discussed in this chapter apply to other layers

Functions of the Data Link Layer

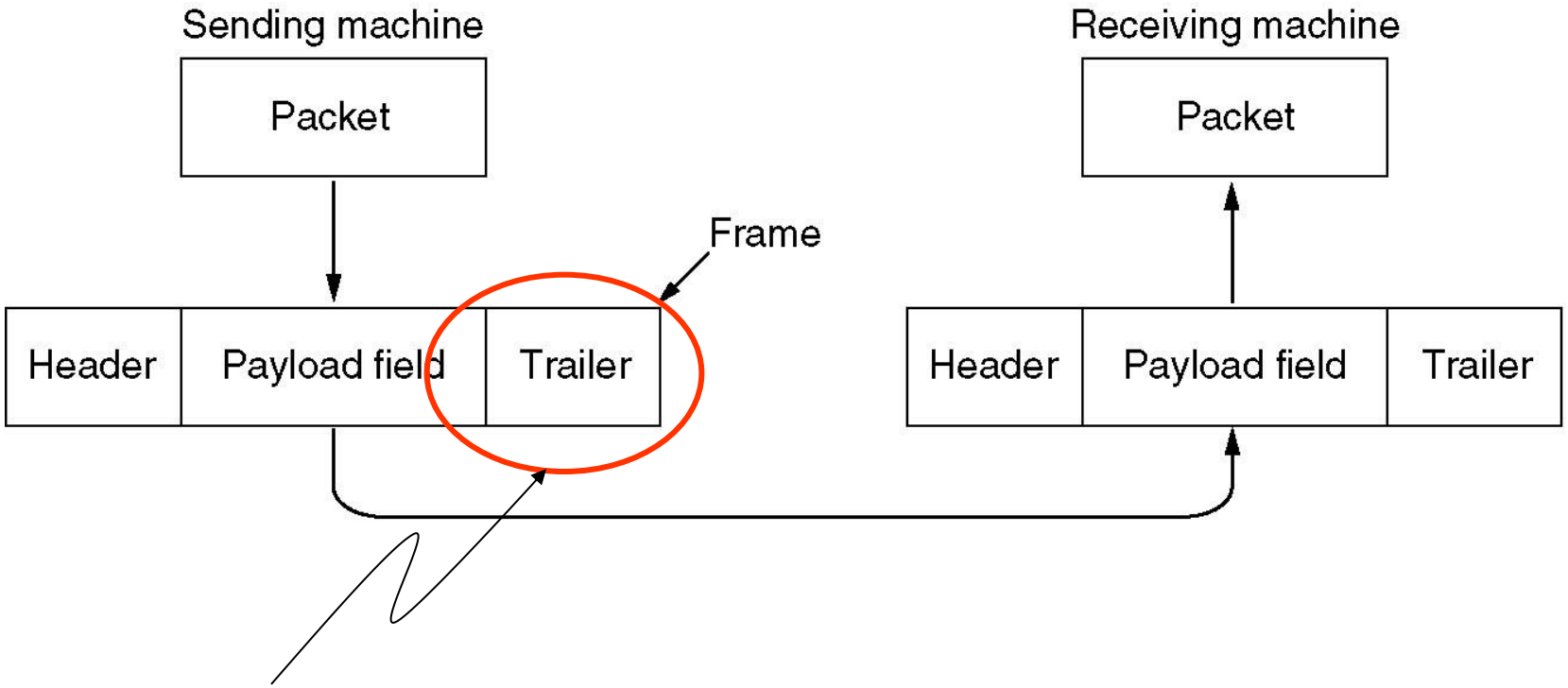
- Provide service interface to the network layer
- Dealing with transmission errors.
  - Either error free or detectable
- In broadcast networks there are additional problems
  - Co-ordination (e.g. CDMA, token, central arbiter)
  - Local Addressing (within a network)
- Regulating data flow
  - Slow receivers not swamped by fast senders
  - Not always implemented at the datalink layer
  - Provided only up to the interface to network layer
    - Does not guard against packet drop in upper layer

Data Link Layer Design Issues

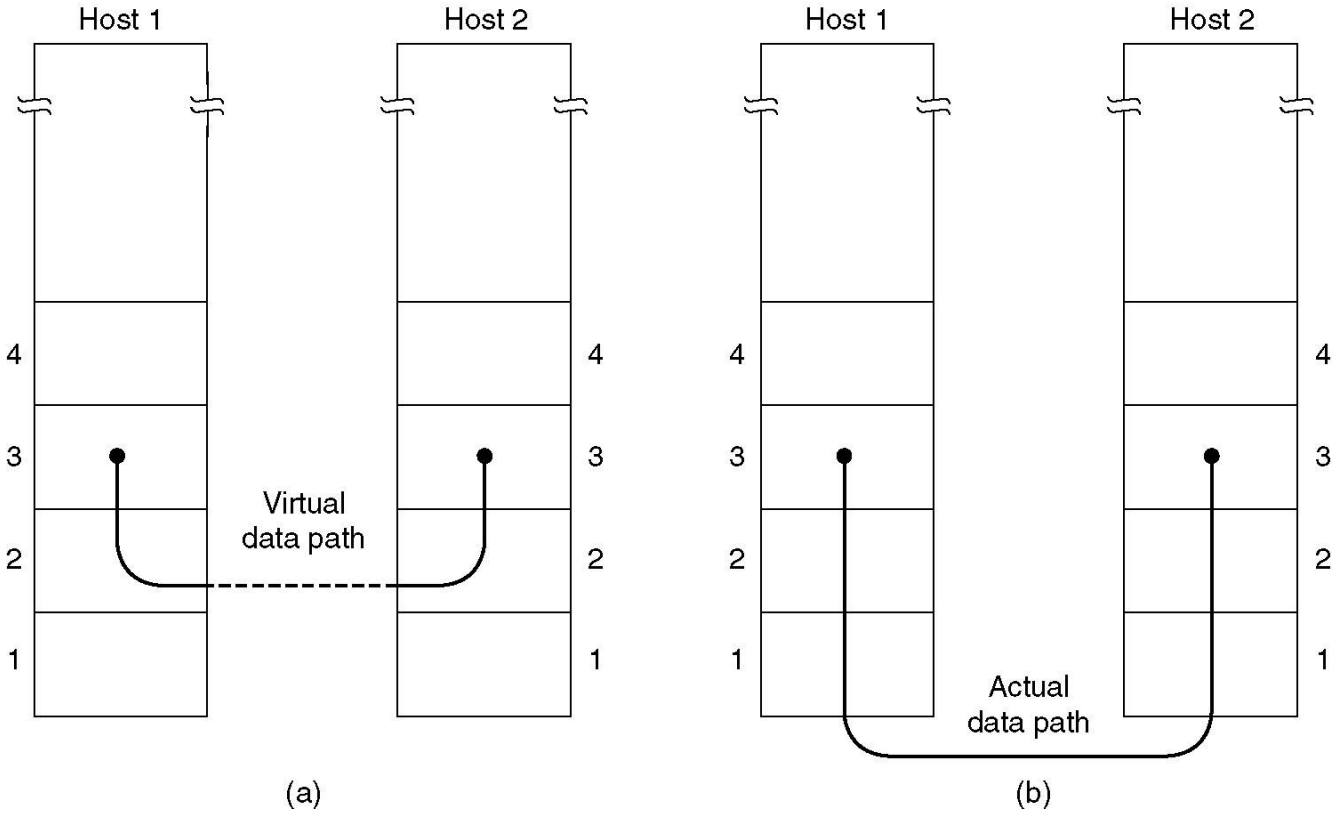
- Services Provided to the Network Layer
  - Framing
  - Error Control
  - Flow Control
  - Local Addressing
- Many of the above functions are also provided at upper layers. E.g.
  - E.g. Error control and flow control is provided at the transport layer
  - The principle is the same in all layers

Relation Between Packets and Frames

Relationship between packets and frames.



Services Provided to Network Layer



Types of Connections

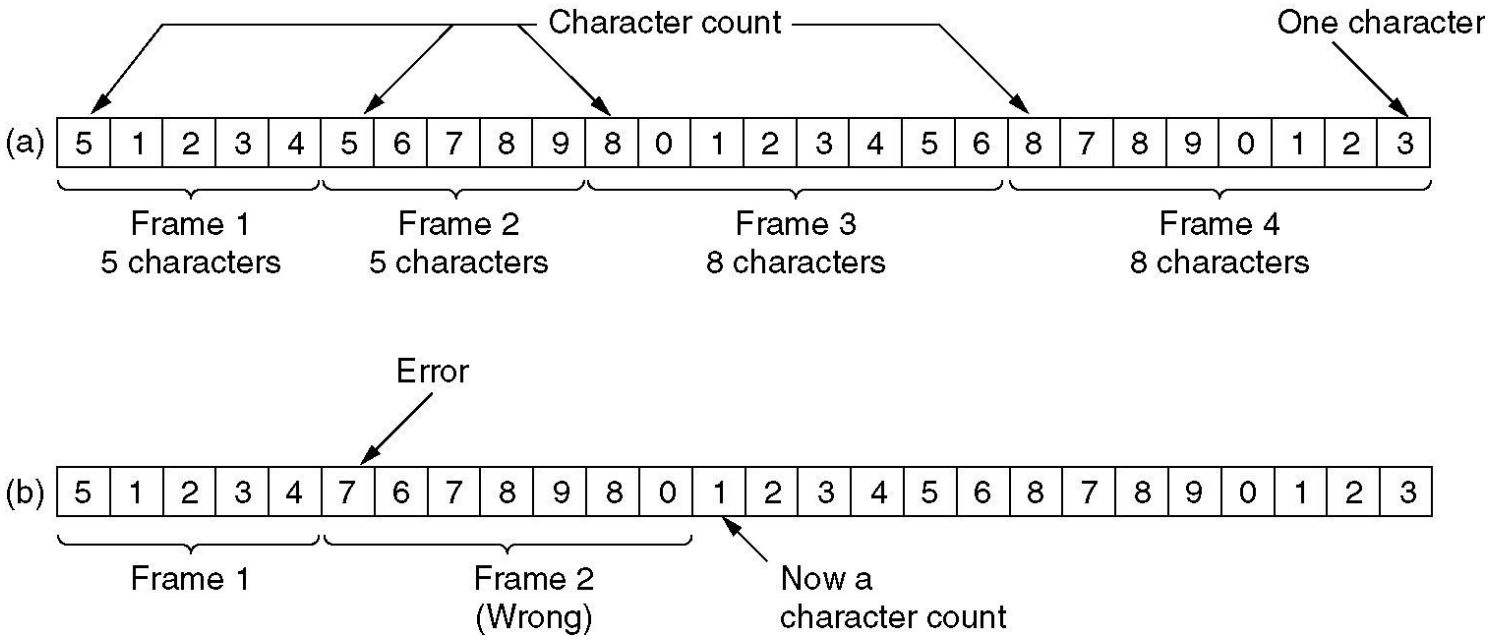
- Unacknowledged connectionless
  - Usually on reliable links (e.g fiber).
  - Extra weight and complexity of acknowledgement not necessary
- Acknowledged connectionless
  - Good for unreliable links* (e.g. wireless)
  - Sub-optimal for reliable channels (e.g. fiber)
  - No need for connection establishment
    - Mobility and simplicity
    - No need to keep state with every peer
  - Can deliver *out of order* packets
- Connection oriented
  - Establish connection prior to sending
  - Information is acknowledged
  - Easy to provide flow control
  - Each packet gets a sequence number
    - In-order delivery
    - Easy to detect lost packet and expedite retransmit (E.g. by sending NACK)
  - PPP, HDLC

Framing

- Framing is breaking up stream of bits into well defined bounded chunks
- Good for reliability
  - Error detection/correction codes
  - Allows for Ack
- Not easy to do (Cannot rely on time gaps)
  - time gaps are not guaranteed by network
  - Time gaps may change while traveling due physical or logical condition
- We'll talk about 3 frame demarcation methods
  - Character count
  - Flag bytes with byte stuffing
  - Starting and ending flags with bit stuffing
  - Physical layer encoding violations (we will not talk about it)

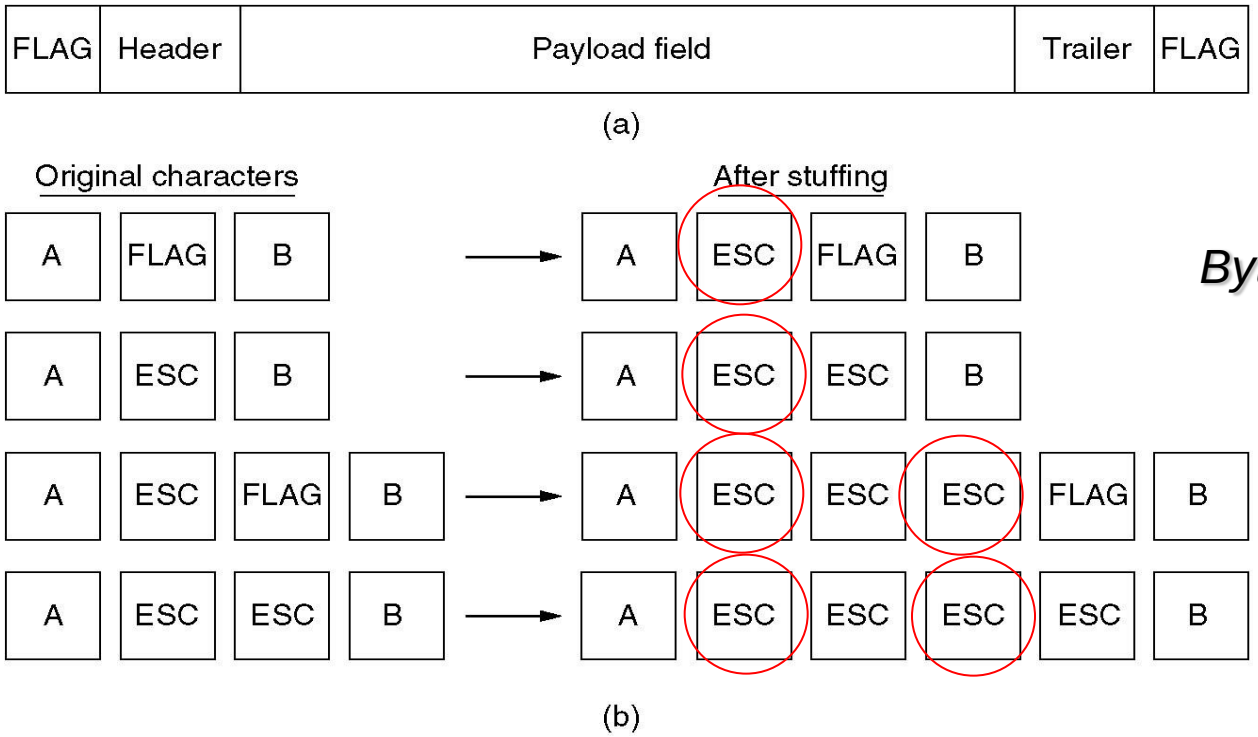
Framing Using Character Count

- Idea
  - Put the character count in the header of each frame



A character stream. (a) Without errors. (b) With one error.

Framing Using Flag Byte



Byte Stuffing

Flag is called Frame Sync Sequence (FSS) as in HDLC

(a) A frame delimited by FLAG bytes.

I.e. Each frame is started and ended by a FLAG byte

The Start and end FLAG bytes may be identical

Why it is better to have a FLAG byte at the beginning and the end of each frame instead of just one FLAG byte at the beginning of each frame?

(b) Four examples of byte sequences before and after stuffing.

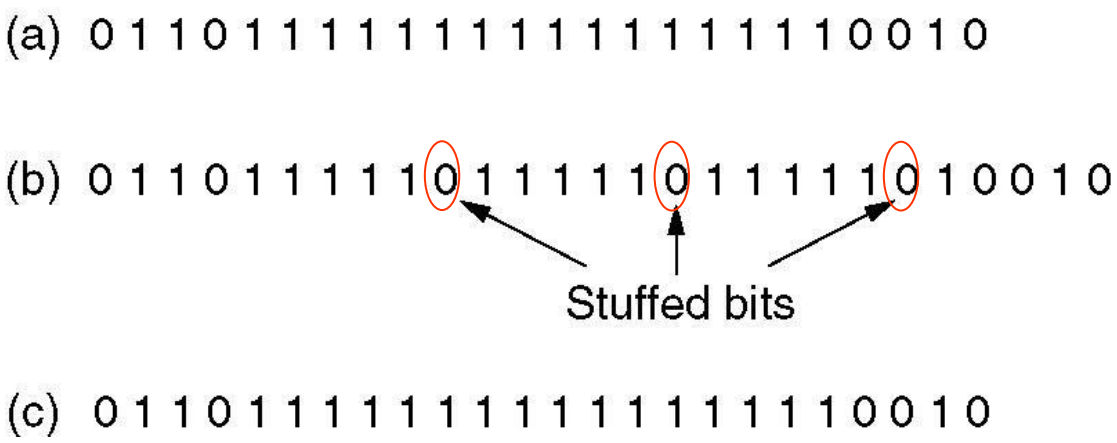
(c) Insert ESC before **each** *accidental* FLAG in data stream

(d) Insert ESC before **each** *accidental* ESC in data stream

(e) A single FLAG is a flag. Double ESC is a single ESC

What is the disadvantage of byte stuffing?

Framing Using Bit Stuffing



- Idea
  - Add (***stuff***) a bit so that the flag sequence is never transmitted inside the payload data
  - Allow the receiver to detect the "***stuffed***" bit and remove it to restore data to its original value
- Flag is 01111110
- When sender sees a sequence of 5 1's it stuffs **0** into the stream
- When receiver sees 5 1's followed by **0**, it *destuffs* the **0**
- Hence we can ***never*** get 01111110 in the data stream
- If stream contains 01111110, it is transmitted as 011111**0**10
- When receiver sees "011111**0**", it removes **0** to get "011111"

Error Control

- Error control repairs frames that are received in error
  - Requires errors to be detected at the receiver
  - Typically retransmit the unacknowledged frames
  - Timer protects against lost acknowledgements
- Detecting errors and retransmissions are next topics.

Flow Control

- Prevents a fast sender from out-pacing a slow receiver
  - Receiver gives feedback on the data it can accept
  - Rare in the Link layer as NICs run at “wire speed”
    - Receiver can take data as fast as it can be sent

- Flow control is a topic in the Link and Transport layers

Error Detection and Correction

kol el correction algorithms 2ayma 3la concept wa7d, enk tzwd r redundant bits, w menhom t3ml check 34an te3rf mkan el error fen bzbt w t7lo.

fa bnb3t m bits of the message and add to them r bits.
total message size = m (message) + r (redundant bits).

- Two main strategies
  - Error-Correcting Codes (FEC)
  - Error-Detecting Codes
- An error is a bit the value of which is reversed
  - Error sometimes is referred to as "corruption"
- Errors are usually bursty
  - Bursty errors cause less packets to be corrupted than random error
  - Much harder to correct because *many* bits are corrupted.
- FEC good for low reliability (e.g. wireless) because retransmission is frequent and may contain error itself
- Error detection is good for reliable links because it is cheaper to re-transmit the rare corrupted frames

- **What is the basic idea?**
  - ***Agree on valid symbols***
  - ***Add check (redundant) bits***
- ***Why do we need redundant bits?***

Original Hamming Method

- Use power of 2 bit position as check bits (1, 2, 4, 8, 16)
- All other bit position are message bits
- The parity bit at position 2^k checks bits in positions having bit k set in their binary representation.
- Conversely, for instance, bit 13, i.e. 1101, is checked by bits 1000 = 8, 0100=4 and 0001= 1.

Error Bounds – Hamming distance

Code turns data of n bits into codewords of n+k bits

Hamming distance is the minimum bit flips to turn one valid codeword into any other valid one.

- Example with 4 codewords of 10 bits (n=2, k=8):
  - 0000000000, 0000011111, 1111100000, and 1111111111
  - Hamming distance is 5

el fekra kolaha baa enk lama t5tar words, ykon el distance benhom kber, mtro7sh t3ml 0000, 0001, keda el d = 1 , fe el e = 0.

Bounds for a code with distance:

- 2d+1 – can correct d errors (e.g., 2 errors above)
- d+1 – can detect d errors (e.g., 4 errors above)

errors to correct = $\frac{\text{hamming distance} - 1}{2}$ = 5 - 1 / 2 = 2

errors to detect = hamming distance - 1 = 5 - 1 = 4.

Hamming Method

- **For 1-bit error-correction, message m, number of redundant r bits will be calculated as follows:**
  - **$(m+r+1) <= 2^r$**
  - Let m=64. Then we need:
    $r+65 <= 2^r$
    Remember **powers of 2**
  - i.e. r >= 7

lw 3auzen ne3ml 1 bit error correction bl hamming method.
e7na 2olna en bn7ot check bits, fe el 2^k, w lazm a2dr a3ml cover le kol el message, y3ny lw 3ndy msg toulha 64 bits el amakn el mfrod tkon
2^0, 2^1, 2^2,2^3,2^4,2^5
1 , 2, 4, 8,16,32,64

fa dol 3adadhom 7.

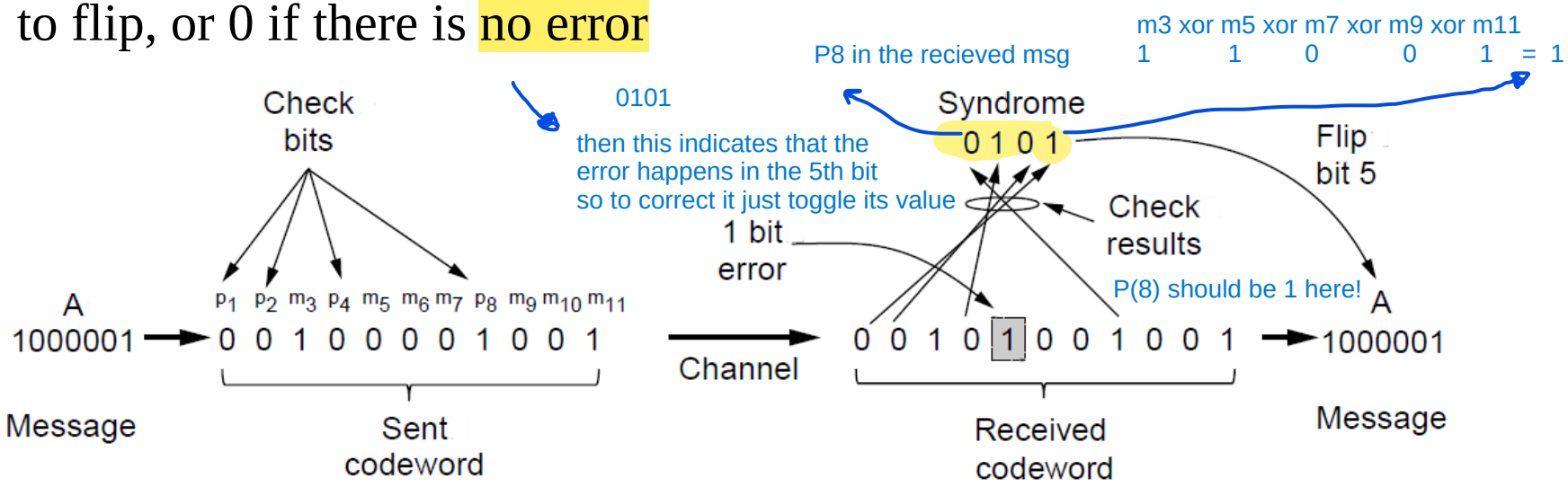
tb hal ana baa kol mara h3od a7sbha manually keda?
alak laa feh equation a2dr ast5dmha 34an a7sb el klam da.
alak 34an te2dr t3ml correction lazm (m+r+1) tkon asghar mn aw btsawy 2^r

fa hena
m = size of the message = 64, r -> # of redundant bits, which is unknown.
so (64 + 1 + r) <= 2^r

try and error till you find the correct r.
you will find here that 7 is the least correct answer.
w 5ly balk enna dayman bn7awl ndwr 3la a2all 3addadd momken, mtro7sh t7ot 100 masln, ma aked 2^100 > 165, lakn enta keda btday3 bits 3la el fady.

Error Correction – Hamming code

- Hamming code gives a simple way to add check bits and correct up to a single bit error:
  - Check bits are parity over subsets of the codeword
  - Re-computing the parity sums (syndrome) gives the position of the error to flip, or 0 if there is no error



(11, 7) Hamming code adds 4 check bits and can correct 1 error

Even parity: P1 (m3,m5,m7,m9,m11)=0 – P2 (m3,m6,m7,m10,m11)= 0 – P4 (m5,m6,m7)=0 – P8 (m9,m10,m11)=1

Hamming Code to Correct Burst Errors

Char.	ASCII	Check bits
H	1001000	00110010000
a	1100001	10111001001
m	1101101	11101010101
m	1101101	11101010101
i	1101001	01101011001
n	1101110	01101010110
g	1100111	01111001111
	0100000	10011000000
c	1100011	11111000011
o	1101111	10101011111
d	1100100	11111001100
e	1100101	00111000101

Order of bit transmission

- Distribute the burst over multiple code words by sending columns first
- Let the burst error size be *k*
- Arrange every *k* codeword in a matrix
- Send the *k* code words one *column* after the other
- When the entire matrix is received, reconstruct the matrix
- Because burst size is at most *k*, then there is at most ***one*** error per code word

Error-Detecting Codes

- Better for reliable channels
- Detect an error and re-transmit
- Far less redundancy bits.

Error Detection – Parity (1)

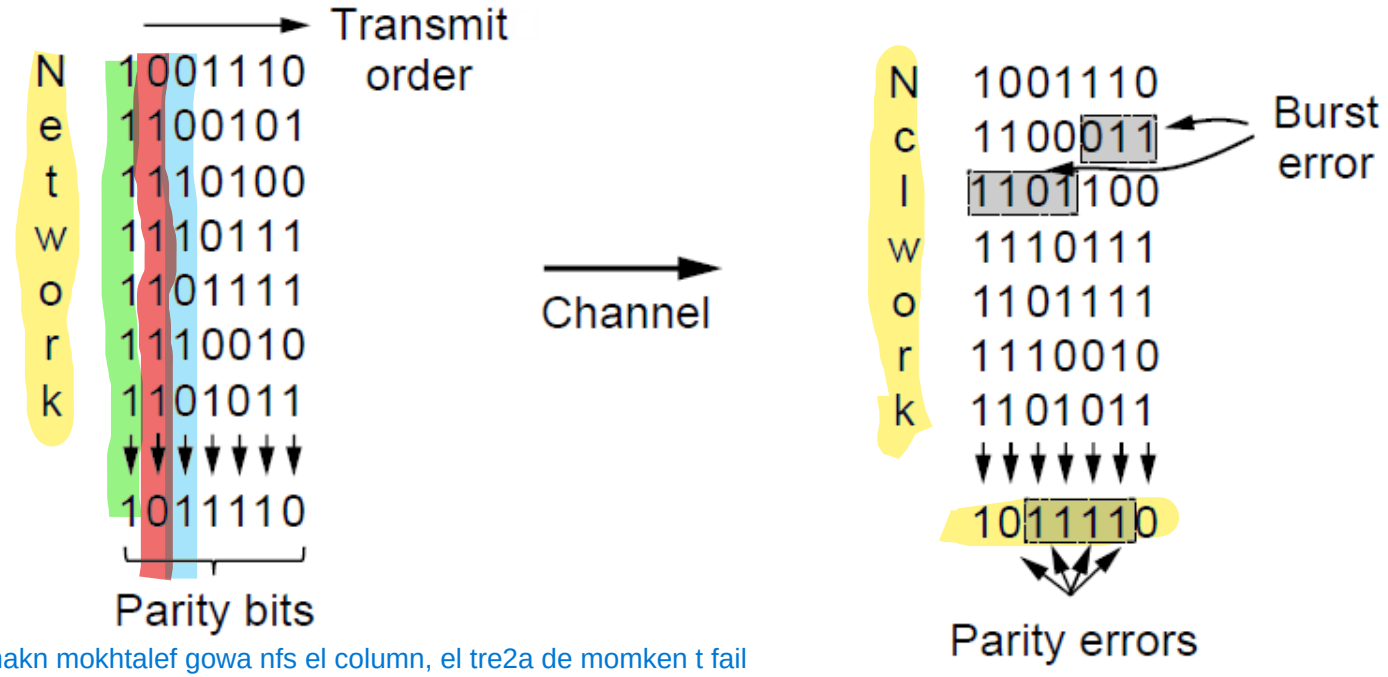
- Parity bit is added as the modulo 2 sum of data bits
  - Equivalent to XOR; this is even parity
  - Ex: 1110000 → 11100001
  - Detection checks if the sum is wrong (an error)

- Simple way to detect an *odd* number of errors
  - Ex: 1 error, 11100101; detected, sum is wrong
  - Ex: 3 errors, 11011001; detected sum is wrong
  - Ex: 2 errors, 1110110 ; *not detected*, sum is right!
  - Error can also be in the parity bit itself
  - Random errors are detected with probability ½

Error Detection – Parity (2)

- Interleaving of N parity bits detects burst errors up to N
  - Each parity sum is made over non-adjacent bits
  - An even burst of up to N errors will not cause it to fail

fl awl bnb2a 3arfen 7agm el word 3ndy kam -> N
fa ay packet hb3tha 7agmaha lazm tkon N
fa b8d el nazar b2a 3n hb3t kam message, ana h7otohom fo2 b3d, w for each column, I will have a parity bit representing it, so # of parity bits used = N



lw 7asl error bs fe amakn mokhtalef gowa nfs el column, el tre2a de momken t fail
00 10
01 --> 11 -> it wil consider it correct, even when there are only N errors occured.

Parity Error-Detecting Codes

- Compare Parity bits with Hamming error correction
- Suppose the error rate is 10^{-6}
- Hamming error correction
  - message size is 1000 bits and we send a block of 1000 messages
  - Using $m + r + 1 \leq 2^r$, we need 10 redundant bits per message ($2^{10}=1024$).
  - Single bit error correction 10kbits per Mbits
- Parity Bits error detection + retransmission
  - Single bit burst error ***correction*** overhead is 2001 bits per Mbits (*how did we calculate this number?)*

Parity Error-Detecting Codes

- 1 M of data needs 1,000 check bits.
- Once every 1000 blocks (1 M), 1 block needs to be re-transmitted (extra 1001)

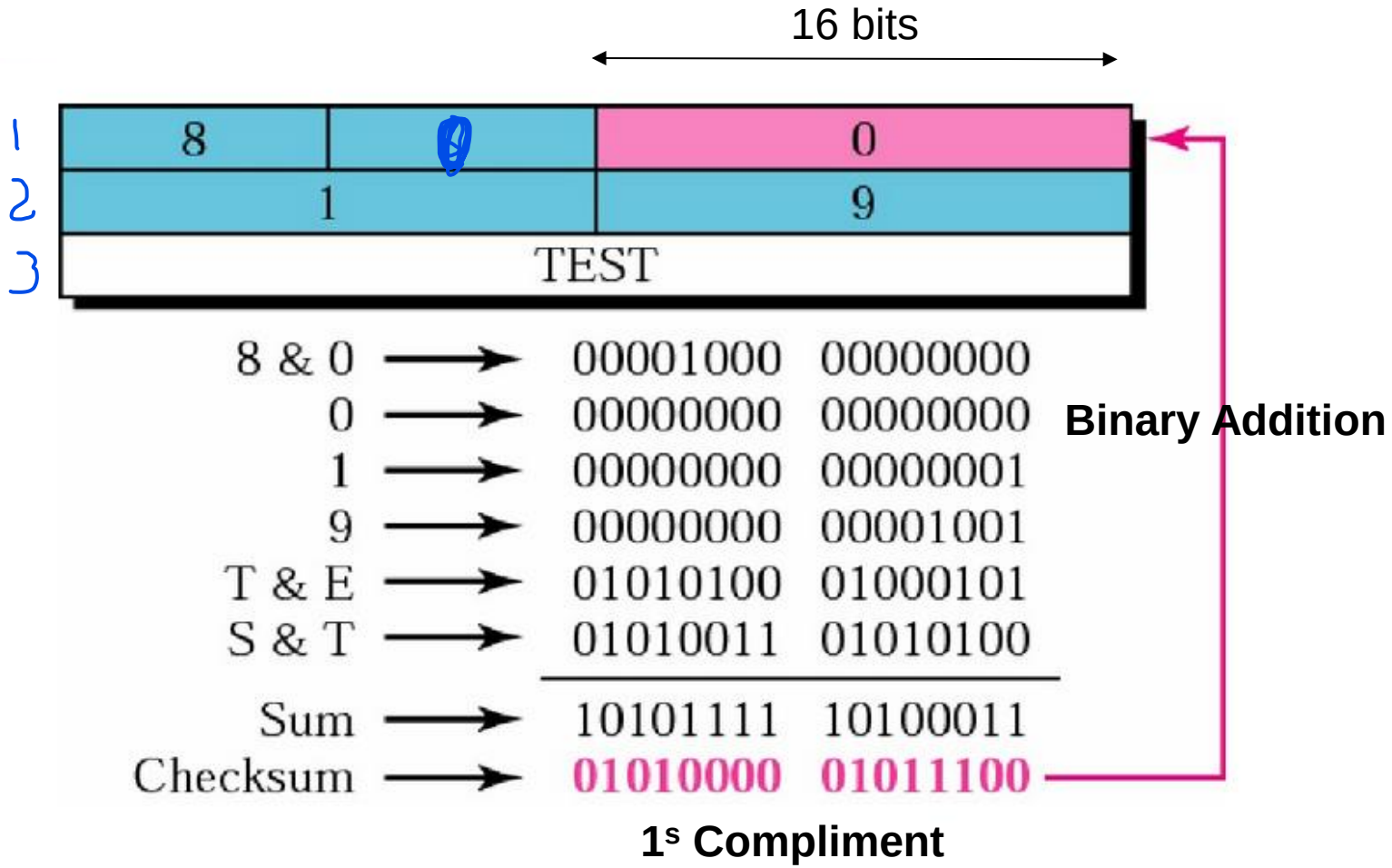
Error Detection – Checksums

- Checksum treats data as N-bit words and adds N check bits that are the modulo 2^N sum of the words
  - Ex: Internet 16-bit 1s complement checksum
  - Simply Binary Add your N-bit words and your checksum is 1s complement of the result

- Properties:
  - Improved error detection over parity bits
  - Detects bursts up to N errors
  - Detects random errors with probability $1-2^N$
  - Vulnerable to systematic errors, e.g., added zeros

Error Detection – Checksums

Example of checksum calculation



Cyclic Redundancy Check Idea

- We all know that *x* divides *y* if remainder of *y/x* is 0
- Transmitter and receiver agree on **generating polynomial** *G(x)*
  - Both high and low order bits of *G(x)* must be **1**.
  - Frame must be longer than *G(x)* ⇒The order of *M(x)* is larger than *G(x)*
- Represent the message (i.e the frame) by the polynomial *M(x)*
- Append **checksum** bits to the message resulting in the polynomial *T(x)*. (*T(x)* contains both *M(x)* and the **checksum** bits)
- Checksum bits are calculated such that *G(x)* divides *T(x)*
- Transmit the frame corresponding to *T(x)* to the receiver
- At the receiver, compute the *received T(x)/G(x)*
- If *G(x)* does ***not*** divide the received *T(x)*, then there is an ***error***

How to Calculate Transmit Frame

- Given $G(x)$ and $M(x)$
- Let r be the degree of $G(x)$
  $\Rightarrow$ $G(x)$ has at most $(r+1)$ terms
- Let m be the maximum number of bits in the message
  - m is the size of the payload
- Append r *0*'s to $M(x)$
  - The frame now contains $\boldsymbol{m+r}$ bits
  - The frame corresponds to the polynomial $x^r M(x)$ (*Why?*)
- Divide $x^r M(x)$ by $G(x)$ using modulo 2 arithmetic
- *What is the number of bits in the remainder*?
- Subtract the remainder from $x^r M(x)$
  - Remember that *subtraction* is just **XOR**.
- $T(x)$ is the result of the subtraction.

Error Detection – CRCs

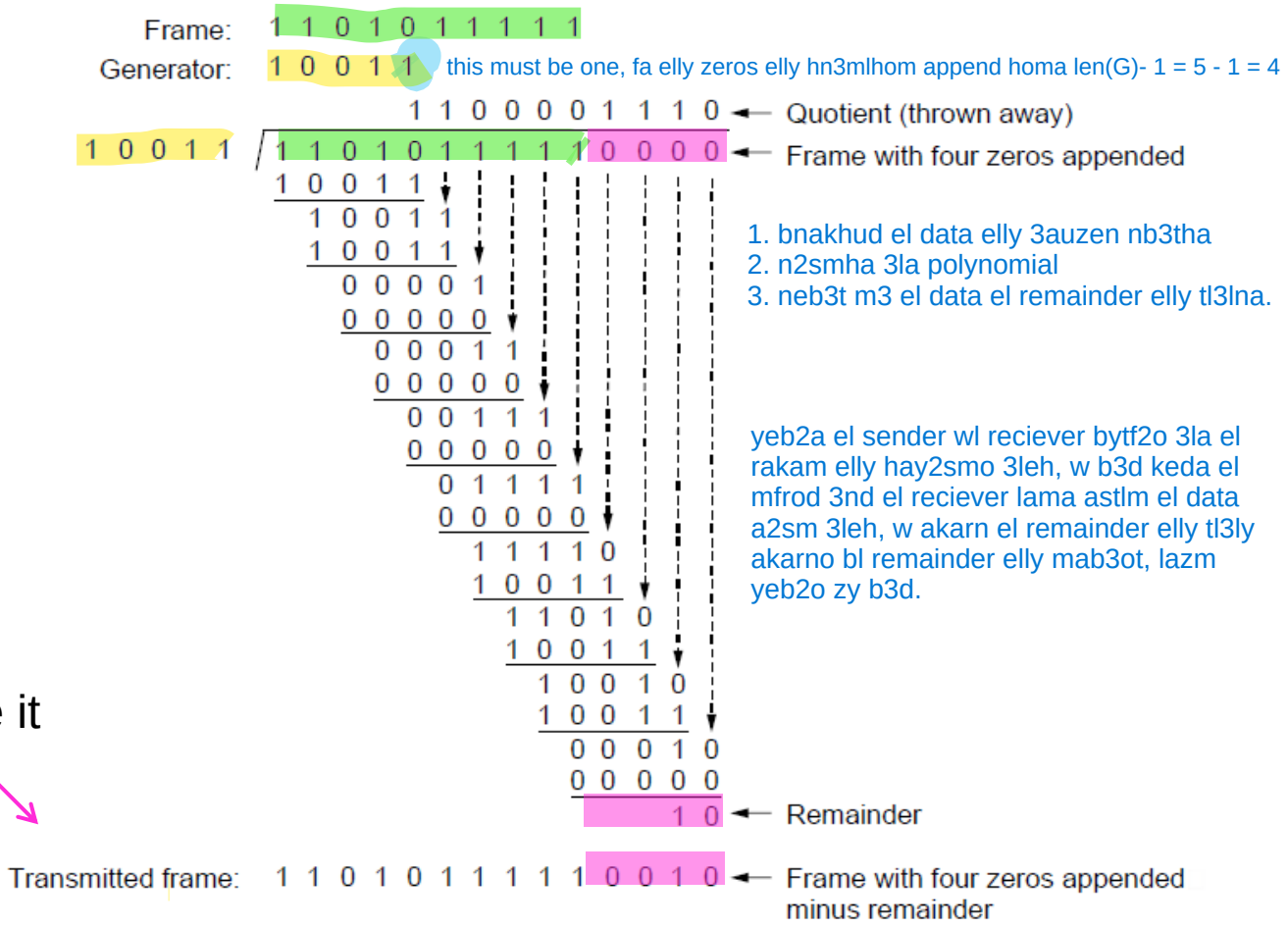
- Adds bits so that transmitted frame viewed as a polynomial is evenly divisible by a generator polynomial

fkeda el r = degree of G = len(generator) - 1 = 5 - 1 = 4.

Start by adding 0s to frame and try dividing

- Add zeros to end
- Simple Xor and shift
- Get the remainder

Offset by any remainder to make it evenly divisible



1. bnakhud el data elly 3auzen nb3tha
2. n2smha 3la polynomial
3. neb3t m3 el data el remainder elly tl3lna.

yeb2a el sender wl reciever bytf2o 3la el rakam elly hay2smo 3leh, w b3d keda el mfrod 3nd el reciever lama astlm el data a2sm 3leh, w akarn el remainder elly tl3ly akarno bl remainder elly mab3ot, lazm yeb2o zy b3d.

Error Detection – CRCs

- Based on standard polynomials:
  - Ex: Ethernet 32-bit CRC

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

11011011011100010000011001000001 <= reversed <= x1x2x3,,,,,x32

  - Computed with simple shift/XOR circuits

- Stronger detection than checksums:
  - E.g., can detect all double bit errors
  - Not vulnerable to systematic errors

Elementary Data Link Protocols

1. An Unrestricted Simplex Protocol (UTOPIA)
2. A Simplex Stop-and-Wait Protocol
3. A Simplex Protocol for a Noisy Channel

- Simplex means:
  - Data can flow in one direction only
  - Control packets can flow in both directions

Variations of protocols described in this section are used in other layers

Basic Assumptions

- Assume physical, data link, and network layers are independent processes that communicate only using messages
- The physical layer does **NOT reorder** packets.
  - In general, physical wires and media do not re-order packets.
  - Reordering usually occurs in virtual channels
- **Assume network layer at endpoints want *reliable* connection-oriented channel**
  - **In order delivery: Packets are always delivered to network layer in order**
  - **No duplicate packets: Never deliver duplicate packets to the network layer**
  - **No packets are lost: Never skip delivering a packet to the network layer.**
  - **No error packets: The payload delivered to the network layer of the receiver is identical to the payload sent by the network layer at the sender**
  - **No deadlock: Packets are delivered in a *finite amount of time*. E.g a packet never gets stuck at the sender or the receiver datalink layer**

    Note: Unbounded wait time (e.g. due to strict priority) is considered finite. Hence for a protocol to be reliable it need not provide bounded time but it has to be finite*)
- Frame header is never given to network layer (to ensure layer independence)
- Assume medium is unreliable. Hence timeout mechanism is used by sender to retransmit.
- Assume timers can be (re)started and stopped at any time
- Assume receiver waits indefinitely and only wakes on events (e.g. packet arrive, received errored packet, timer expired)
- Subtraction and addition of sequence number is assumed to be *circular*
  - We *never* get negative sequence number
- Assume network layer transmitter has infinite supply of data
  - we will remove this assumption later

Protocol Definitions

```
#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 a frame is it? */
  seq_nr seq;                                         /* sequence number */
  seq_nr ack;                                         /* acknowledgement number */
  packet info;                                        /* the network layer packet */
} frame;
```

Frame header

The sequence number of the frame that we are sending in this packet

The sequence number of the frame to be received

Payload

Continued →

Some definitions needed in the protocols to follow.
These are located in the file protocol.h.

Protocol Definitions (ctd.)

```
/* 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
```

Separate timer for each sequence number "k"

Some definitions needed in the protocols to follow. These are located in the file protocol.h.

Circular increment

Link layer environment (2)

- Link layer protocol implementations use library functions
  - See code (`protocol.h`) for more details

Group	Library Function	Description
Network layer	from_network_layer(&packet) to_network_layer(&packet) enable_network_layer() disable_network_layer()	Take a packet from network layer to send Deliver a received packet to network layer Let network cause “ready” events Prevent network “ready” events
Physical layer	from_physical_layer(&frame) to_physical_layer(&frame)	Get an incoming frame from physical layer Pass an outgoing frame to physical layer
Events & timers	wait_for_event(&event) start_timer(seq_nr) stop_timer(seq_nr) start_ack_timer() stop_ack_timer()	Wait for a packet / frame / timer event Start a countdown timer running Stop a countdown timer from running Start the ACK countdown timer Stop the ACK countdown timer

Unrestricted Simplex Protocol

- Perfect world ☺
- Error free communication channel
- Sender has infinite buffer
- Receiver has infinite buffer
- Receiver can process all input infinitely fast
- Sender network layer has infinite packet supply to send
- Propagation delay is negligible

Unrestricted Simplex Protocol

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

•We have not initialized the fields in the header (*why?*)

Simplex Stop-and-Wait Protocol

- Semi-perfect world
- Drop the unrealistic assumption at the receiver. Assume
  - Receiver has *finite buffer*
  - Receiver has *finite processing* capacity
- Error free channel
- Sender has infinite buffer
- Sender Network layer has infinite supply of packets to send
- *What and where error can occur?*

Packet drop at the receiver side

Simplex Stop-and-Wait Protocol

- We do *not need a timeout* mechanism in this protocol (*Why?*)
- We do not need sequence number (*why?*)

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

Simplex Stop-and-Wait Protocol

- We can see from the protocol code that we do *NOT have a timeout* on the sender side. *Why don't we need a timer even though the receiver has a finite buffer and hence packets may be dropped?*
  - Actually, there is *no possibility of packet drop* because the sender will send the next packet ONLY when it receives an ACK from the receiver.
  - Because we assumed perfect channel, the packet will arrive at the receiver and the ACK will arrive at the sender
  - Hence there is no need for timeout at the sender
  - In other words, we overcome the finite CPU and buffer capacity of the receiver by having the ***receiver control the transmission***.
- We do not need a sequence number , *why?*
  - because we do not lose packets
  - Because there is no possibility of packet duplication
  - Because there is no possibility of packet reorder
- *Why doesn't the sender inspect the packet type?*

A Simplex Protocol for a Noisy Channel

- Assume *non-zero* variable *propagation delay* along the channel in *both direction*
- Assume possible frame damage or loss in *both direction*
- Assume errored packets are *detected correctly* (*i.e. perfect error detection)*
- Simple modification to the previous protocol
- Use a **timeout** at the sender
- Sender only transmits next frame if it received *correct ACK* (damaged ACK are discarded)
- **If it does not receive a correct ACK, it times-out and re-transmits the same frame**

A Simplex Protocol for a Noisy Channel

- Assume *non-zero* variable *propagation delay* along the channel in *both direction*
- Assume possible frame damage or loss in *both direction*
- Assume errored packets are *detected correctly* (*i.e. perfect error detection*)
- Simple modification to the previous protocol
- Use a **timeout** at the sender
- Sender only transmits next frame if it received *correct ACK* (damaged ACK are discarded)
- **If it does not receive a correct ACK, it times-out and re-transmits the same frame**
- ***Is this correct?***

A Simplex Protocol for a Noisy Channel

- Assume *non-zero* variable *propagation delay* along the channel in *both direction*
- Assume possible frame damage or loss in *both direction*
- Assume errored packets are *detected correctly* (*i.e. perfect error detection*)
- Simple modification to the previous protocol
- Use a **timeout** at the sender
- Sender only transmits next frame if it received *correct ACK* (damaged ACK are discarded)
- **If it does not receive a correct ACK, it times-out and re-transmits the same frame**
- ***Is this correct?***

Duplicate Frame if ACK is lost

A Simplex Protocol for a Noisy Channel

- Assume *non-zero* variable *propagation delay* along the channel in *both direction*
- Assume possible frame damage or loss in *both direction*
- Assume errored packets are *detected correctly*
- Assume that the propagation delay is greater than zero
- Simple modification to the previous protocol
- Use a timeout at the sender
- Sender only transmits next frame if it received *correct ACK* (damaged ACK are discarded)
- If it does not receive a correct ACK, it times-out and re-transmits the same frame
- *Is this correct?* ***Duplicate Frame if ACK is lost***
- We need a sequence number (*what size?*)
- Protocols where sender waits to ACK from receiver and retransmits after time out are sometimes called
  - PAR: Positive Acknowledge with Retransmit
  - **ARQ: Automatic Repeat reQuest**
  - See for example RFC3366: Advice to Link Designers on Link ARQ
- Hence the rest of the protocols in this chapter belong to the class of protocols known as ARQ

A Simplex Protocol for a Noisy Channel

A positive acknowledgement with retransmission protocol.

*Why does the sender need to check the sequence number in **s.ack**?*

I.e. Can the sender just send the next frame as soon as it receives an ACK, assuming that the physical layer drops any corrupted packet?

- Assume that an errored frame generates the event "checksum_err". We will ignore this event.
- This is similar to assuming that physical layer discards errored packets

```
/* 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 == 1- 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 */
          }
      }
  }
}
```

Circular increment

Continued →

A Simplex Protocol for a Noisy Channel (ctd.)

```
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 =      frame_expected;         /* tell which frame is being acked */
          to_physical_layer(&s);               /* send acknowledgement */
      }
  }
}
```

- Assume that an errored frame generates the event "checksum_err". We will ignore this event.
- This is similar to assuming that physical layer discards errored packets

- Receiver sends ACK for next expected frame
- Receiver sends ACK ***every time*** it receives ANY packet

*•Suppose that the receiver does NOT send an ACK unless it receives a packet with the **expected sequence number**, will the protocol work correctly?*

Stop-and-Wait ARQ

Cases of Operations:

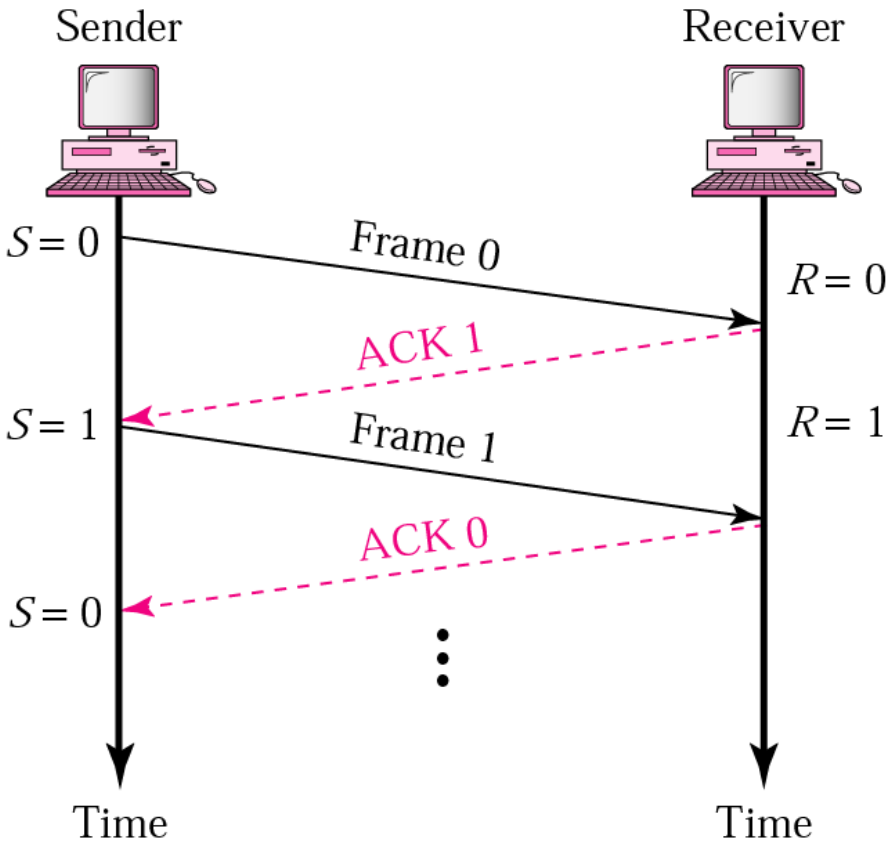
1. *Normal operation*
2. *The frame is lost*
3. *The Acknowledgment (ACK) is lost*
4. *The Ack is delayed*

S&W + Time out + Seq. No. + Ack No.

Stop-and-Wait ARQ

1. Normal operation

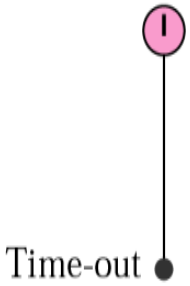
- *The sender will not send the next frame until it is sure that the current one is correctly receive*
- *sequence number is necessary to check for duplicated frames*

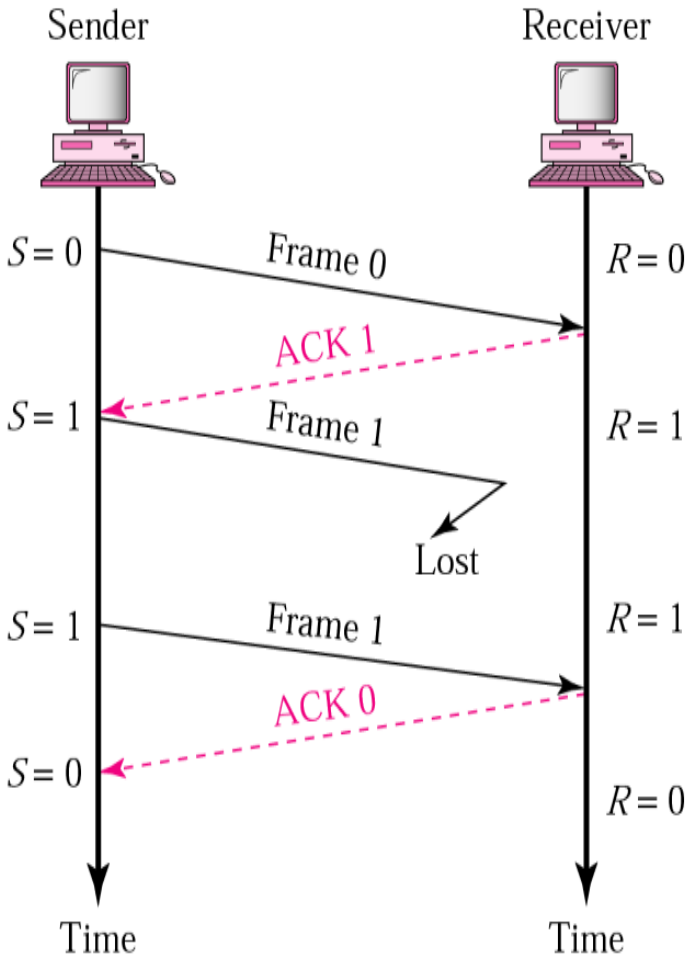


Stop and Wait ARQ

2. Lost or damaged frame

- *A damage or lost frame treated by the same manner by the receiver.*
- *No NACK when frame is corrupted / duplicate*

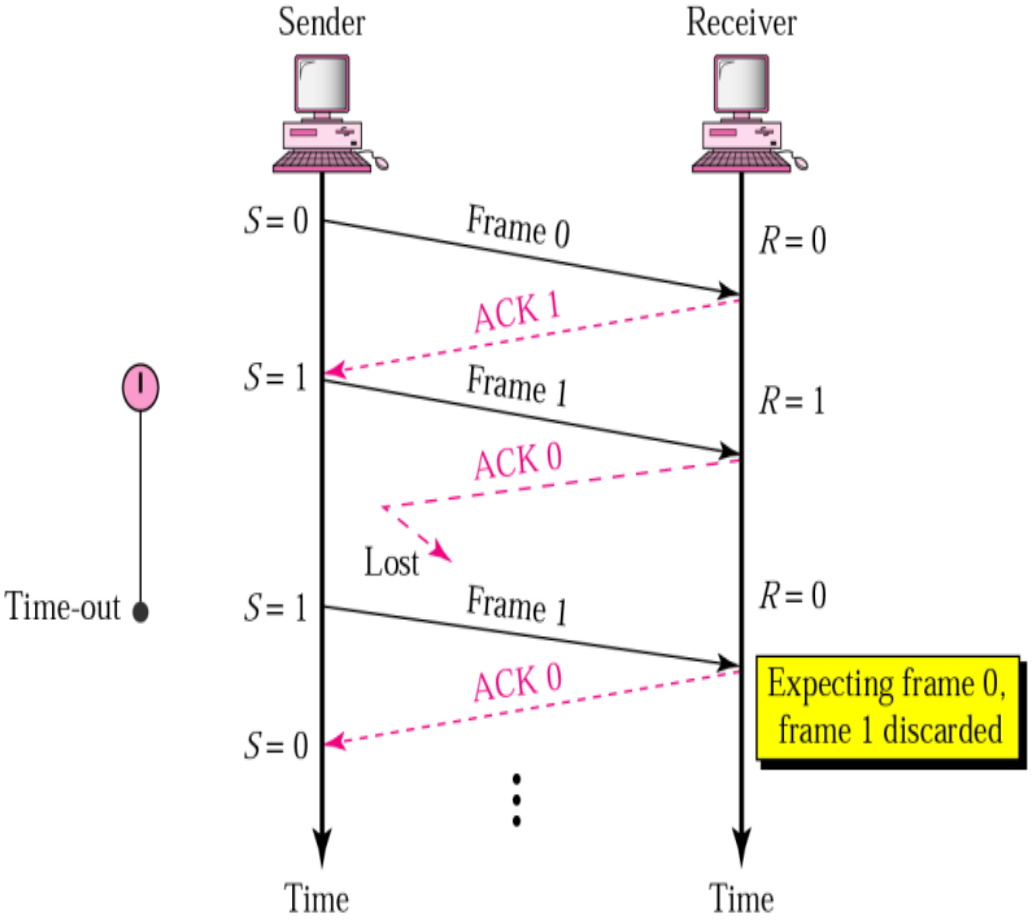




Stop-and-Wait ARQ

3. Lost ACK frame

- *Importance of frame numbering*



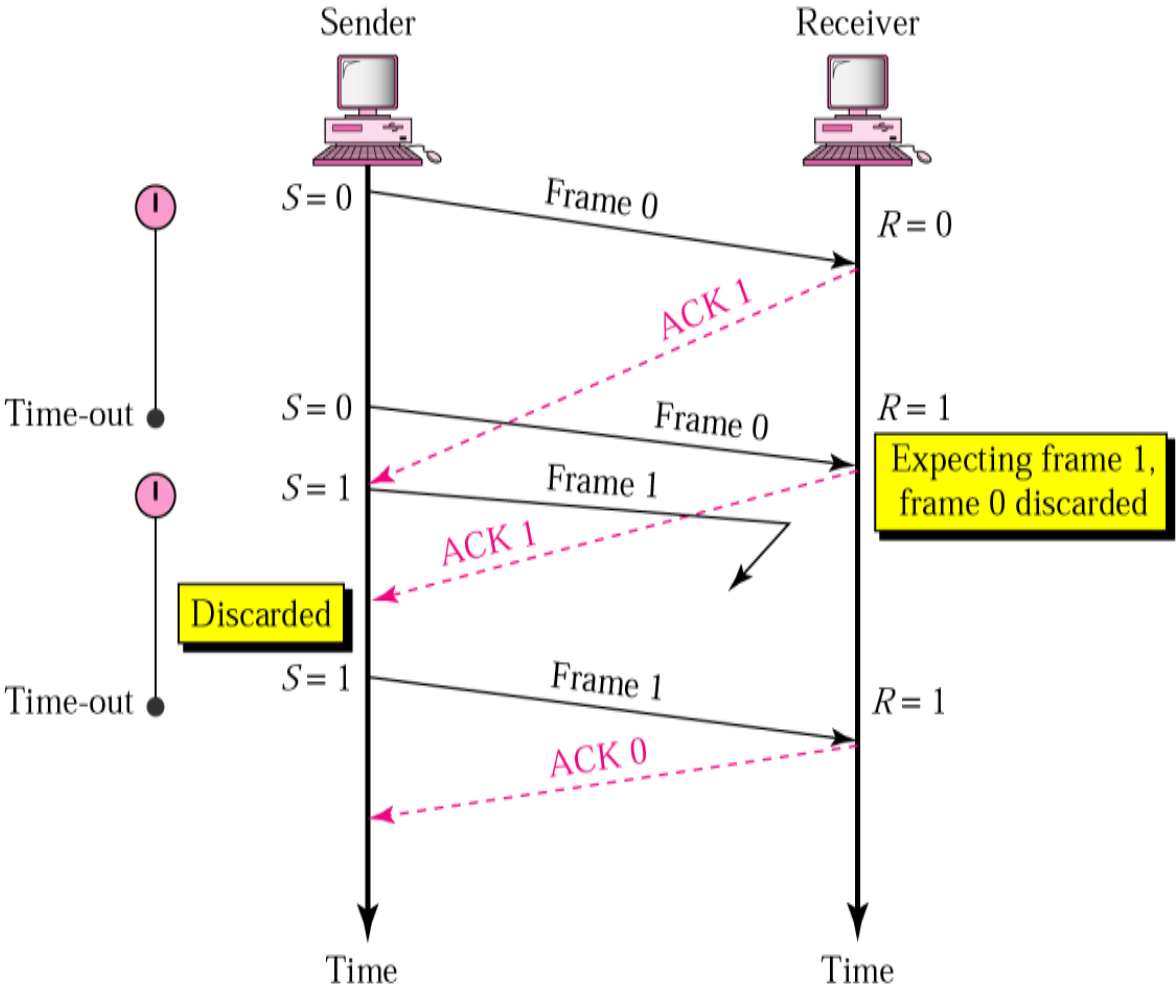
Note

In Stop and-Wait ARQ, numbering frames prevents the retaining of duplicate frames.

Stop-and-Wait ARQ

4. Delayed ACK and lost frame

- *Importance of frame numbering*

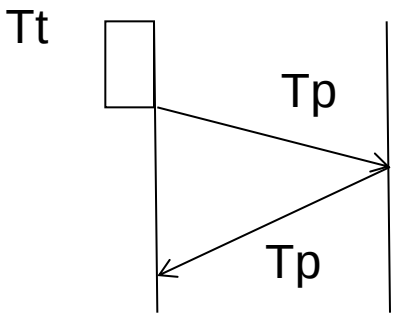


Note

Numbered acknowledgments are needed if an acknowledgment is delayed and the next frame is lost.

S&W Efficiency

- *Efficiency = useful time / total time*
  = Tt / (Tt + 2 Tp)
  = 1 / (1 + (2 Tp/ Tt)) = 1/ (1+2 a) a= Tp/Tt
  = 1/(1+2*(d/v)*(B/L))



- d = distance between source and receiver, v = velocity
- *B = Bandwidth*
- *Throughput : no. of bits sent / sec = L / (Tt + 2Tp)*
  Where L is the packet length
- *Throughput is called Effective Bandwidth or Bandwidth utilization*
- *Effective BW = (L *(B/B)/ (Tt+ 2Tp) = [Tt/Tt+2Tp] * BW*
  *= efficiency*B (Tt=L/B)*

S&W Efficiency

- *Example Tt = 1 msec, Tp = 1 msec , B= 6Mbps,*

  Then Efficiency is 33.3%

  Throughput = 2Mbps

  Tt = 2msec, Tp= 1 msec , Efficiency is 50%,

  Throughput =3Mbps

- *For efficiency >= 50% , then Tt >= 2Tp*
  - *Then (L/B) > = 2Tp then* ***L should be >= 2*Tp*BW***

S&W Efficiency

- In case of **S&W** protocol is only sending only one packet irrespective of the link bandwidth

Example: Tt= 1 msec, Tp = 1.5 msec

a= Tp /Tt =1.5

S&W Efficiency = 1/ (1+2a) = 25%

Retransmission packets in S&W ARQ protocol

- If P is the probability of loss packets
- If we want to transmit n packets then we will transmit the following:

$$n + n(p) + (n(p))(p) + (np\ p)\ p + \ldots.$$

$$n(1 + p + p^2 + p^3 + p^4 + \ldots..)$$

$$= n(1 / (1-p))$$

- Ex: if n = 400 and p= 0.2 then we will transmit 500 packets

A Simplex Protocol for a Noisy Channel Discussion points

- *How can we argue that we only need 1 bit for sequence number?*
- *What anomaly could happen if the timeout at the sender is smaller (even temporarily when sending one packet only) than the round-trip delay?*
- *What is a alternative design for round-trip delay that is larger than sender timeout?*
- *Why does the sender re-send the last packet it sent when it gets duplicate ACK (I.e. what assumption is made)?*
- *Why don't we need a timeout mechanism at the receiver side? In other words, is it possible that the protocol deadlocks and the sender stops sending traffic?*
- *What is main new issue in this protocol?*
- *Why do we put s.ack = 1 –frame_expected*

Piggybacking

- Sending multiple types of packets in the same frame
- Example: Bidirectional channel
  - Receiver sends ACK with data
- Why
  - Saving on bandwidth
  - Saving of packet processing capacity of routers and switches
- Problem: What happens if there is no data to send?
- Use ad hoc mechanisms: e.g.
  - Wait for a short timeout
  - If no new packet arrives after a receiver ack timeout

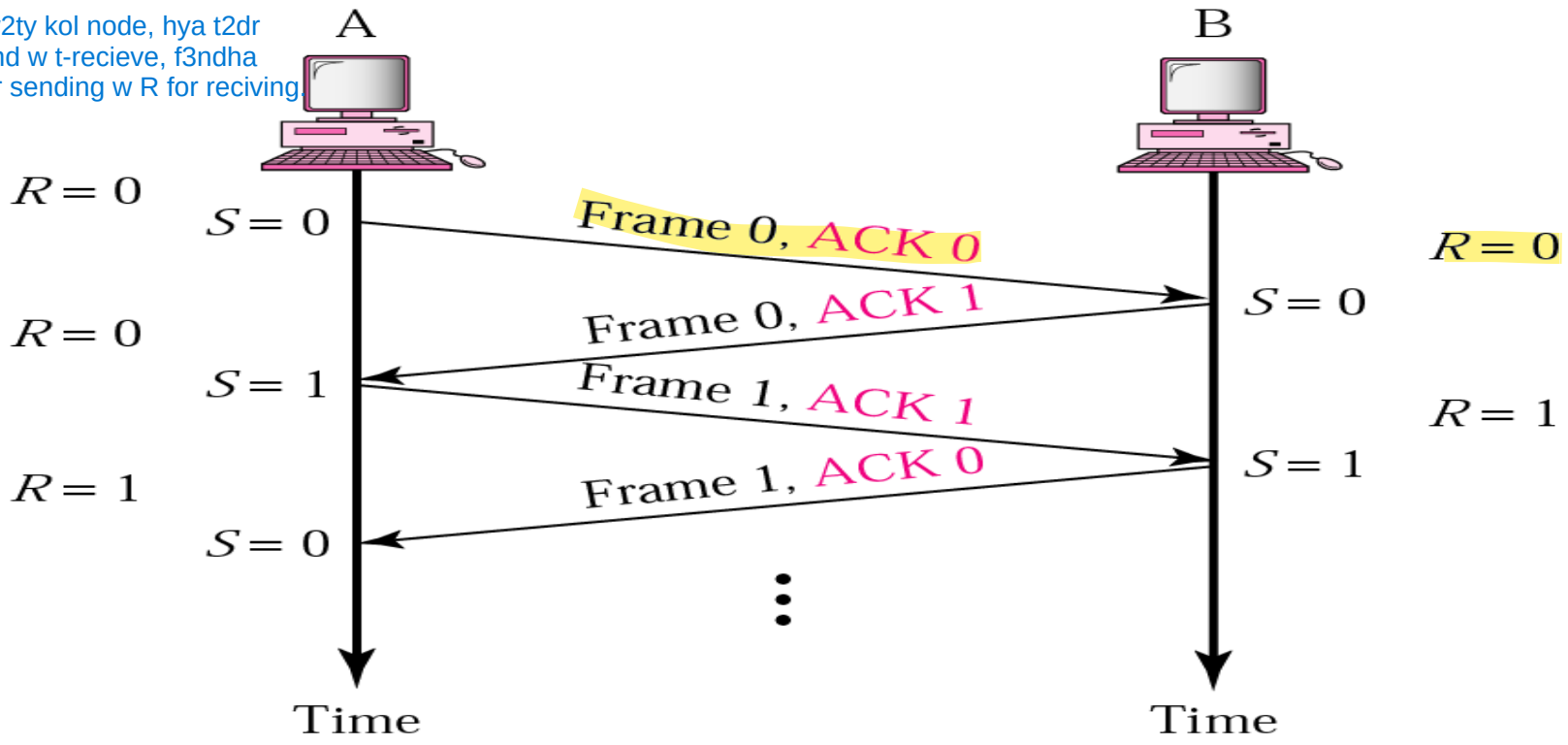
    $\Rightarrow$ Sending a separate acknowledgement frame
  - Timeout should be smaller than other side timeout

Piggybacking (Bidirectional transmission)

Is a method to combine a data frame with an acknowledgment.

It can save bandwidth because data frame and an ACK frame can combined into just one frame

delw2ty kol node, hya t2dr
t-send w t-recieve, f3ndha
S for sending w R for reciving.



Pipelining

Pipelining: A task is begun before the previous task has ended

- ***There is no pipelining in stop and wait ARQ*** *because we need to wait for a frame to reach the destination and be acknowledged before the next frame can be sent*
- *Pipelining improves the efficiency of the transmission*

Link Capacity / Pipelining

- Link Capacity is dependent on BW and Propagation delay
- For a Half duplex :Capacity = BW * Tp
- For a full duplex: Capacity = 2 BW * Tp
- In case of S&W he is only sending one packet irrespective of the link capacity
- With Pipelining :
  - One packet is sent in Tt, How many packets can be send in Tt +2Tp? → (1+2a) packets
- Example: Tt= 1 msec, Tp = 1.5 msec, a= Tp/Tt
  - Efficiency for S&W = 1/ 1+2a= ¼
  - Total time = 4msec, with pipelining you can send 3 more packets

Sliding Window Protocols

- A One-Bit Sliding Window Protocol
- A Protocol Using Go Back N
- A Protocol Using Selective Repeat

- From now on
  - protocols are ***full duplex***
  - ACK is ***piggy-backed*** with data

Sliding window protocol

Sliding window protocols apply Pipelining

- *Sliding window protocols improve the efficiency*
- *multiple frames should be in transition while waiting for ACK. Let more than one frame to be outstanding.*
- *Outstanding frames: frames sent but not acknowledged*
- *We can send up to **W** frames and keep a copy of these frames(outstanding) until the ACKs arrive.*
- *This procedures requires additional feature to be added :sliding window*

Sliding Window concept

Sender maintains window of frames it can send

- Needs to buffer them for possible retransmission
- Window advances with next acknowledgements

el sliding window btb2a fl na7yeten el sender lazm ykon 3ndo sliding window bt3ml 7agten
1. gowaha el frames elly el mfrod ttb3t aw etb3ttet bs lesa m7slhash acknowledge.
2. lama bnege n7rk el window, bn7rkha lama yegy acknowlegde.

Receiver maintains window of frames it can receive

- Needs to keep buffer space for arrivals
- Window advances with in-order arrivals

el reciever baa bardo lazm ykon 3ndo sliding window, bt3ml 7agten brdo
1. bt5zn men el wesel aslun.
2. kol ma wa7d yewsl el sliding window btt7rk.

Sliding Window Protocol

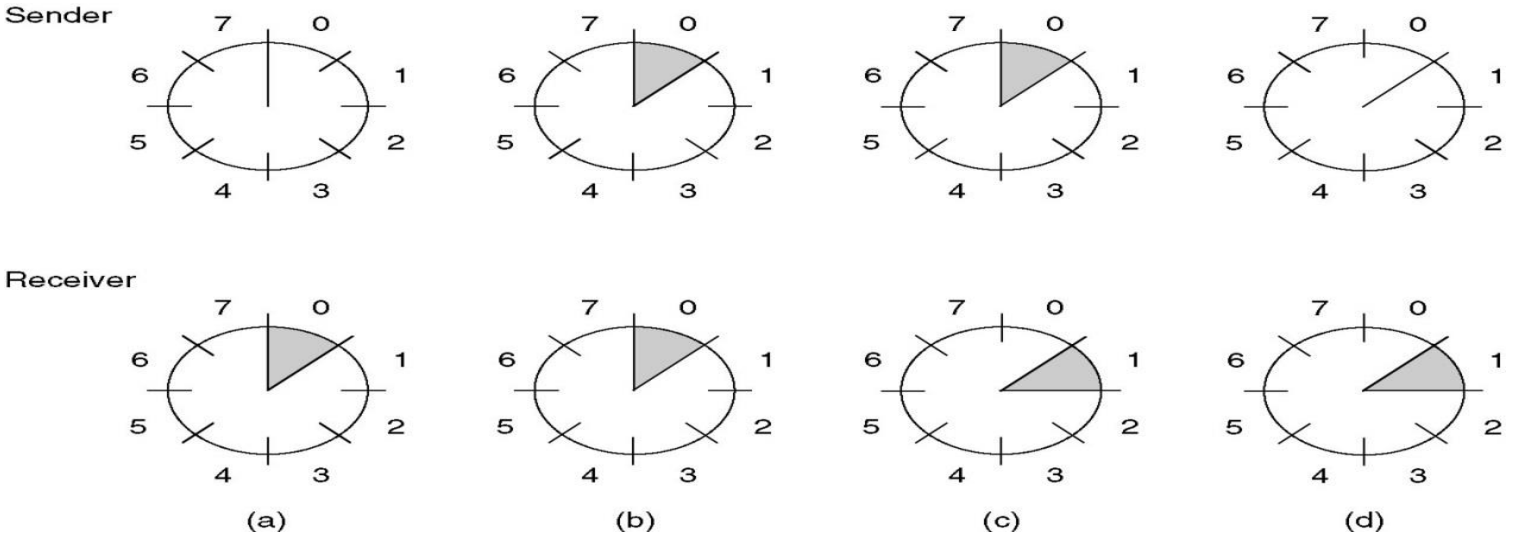
- Sending Window: range of sequence number sender permitted to send or already sent
- Receiving Window: Range of sequence number receiver permitted to receive or partially received
- In general, *Sending window* ≠ *receiving Window*
- Windows need not be fixed in size

- **Sender**
  - Packet from network layer given next highest sequence number
  - ACK matching the lower end of the window advances the lower end
  - ACK may arrive out of order
  - Maximum number of packets transmitted without receiving ACK is *the window size*
  - Retransmit based on timeout (later, re-transmition will be based on other factors)
  - May shutoff link or block network layer if transmit window reaches max size

Sliding Window Protocol

- **Receiver**
  - Accepts only packets within its window (packets outside window discarded)
  - When packet matching the lower end arrives, ACK is sent
  - Packets may arrive ***out of order***
  - Packets are delivered to network layer ***in order***

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.
- *Are packets delivered in order?*
  - Because the receiver window size is 1 packets, packets are always delivered in order

A One-Bit Sliding Window Protocol

```
/* 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 =       frame_expected;                     /* piggybacked ack */
  to_physical_layer(&s);                            /* transmit the frame */
  start_timer(s.seq);                               /* start the timer running */
```

hena mfesh sender w reciever, l2n el etnen shaghalen aknhom sender w reciever.

w hena brdo mfesh pipeline, l2n ana bab3t one frame only.

Continued →

A One-Bit Sliding Window Protocol (ctd.)

```
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 == 1- 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 senderís sequence number */
        }
    }
    s.info = buffer;                        /* construct outbound frame */
    s.seq = next_frame_to_send;             /* insert sequence number into it */
    s.ack =      frame_expected;            /* seq number of last received frame */
    to_physical_layer(&s);                  /* transmit a frame */
    start_timer(s.seq);                     /* start the timer running */
  }
}
```

working as a reciever.

working as a sender.

Sender & Reciever in same time

Can be timeout, chksum_err, or frame_arival ***Ignore all*** *except frame_arrival*

Why do I have two variables, ***frame_expected*** *and* ***next_frame_to_send****? In other words, can I calculate the value of one variable based on the value of the other?*

This is equivalent to saying that ACK is the ***circular****(frame expected-1)*
Circular(x-1) = (x>0) ? (x--) : MAX_SEQ

Assume calling start_timer() again restarts the timer

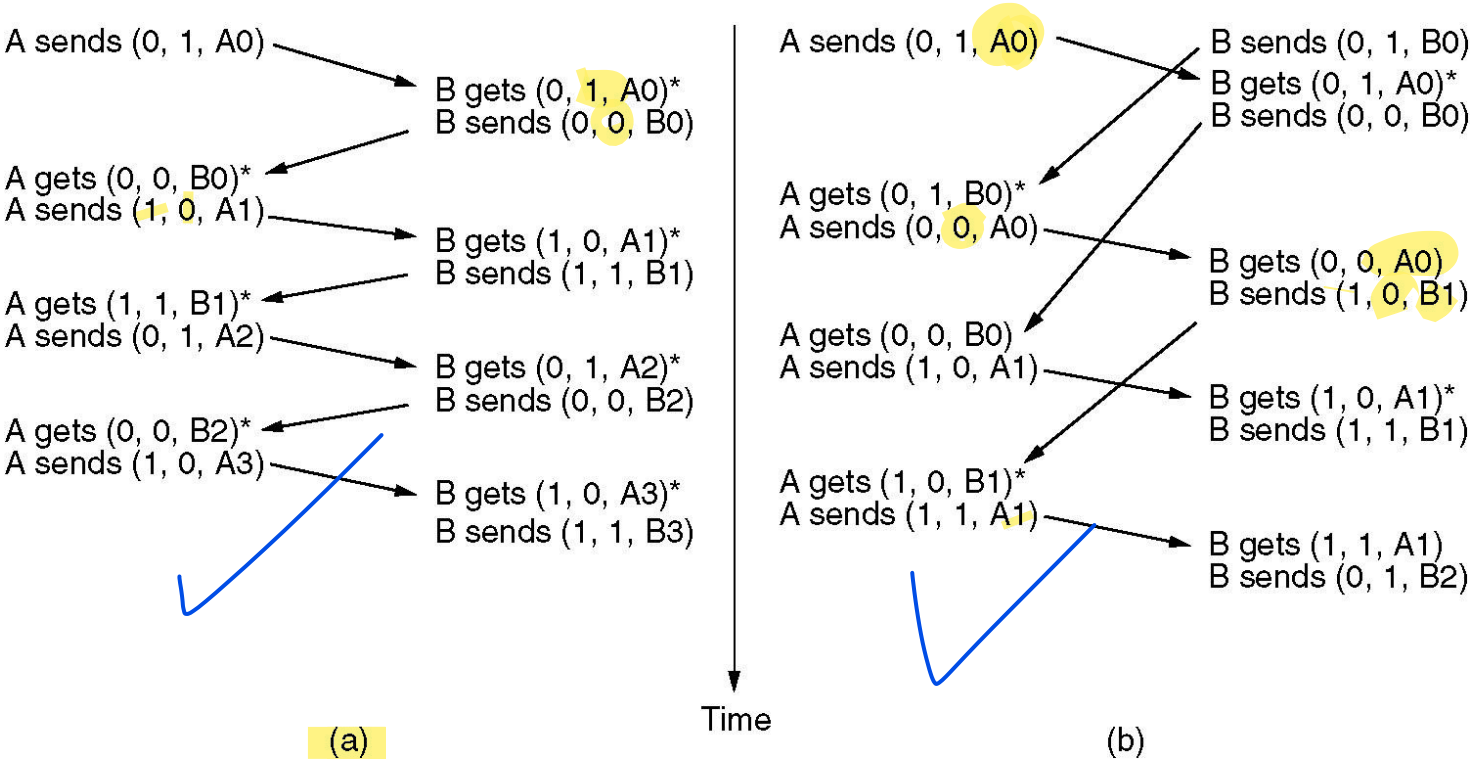
A One-Bit Sliding Window Protocol Correctness

- Protocol is correct
  - In order packet delivery to network layer
  - No packets are skipped
  - No deadlocking. Not stuck packet or keep sending the same packet forever
  - No duplicate packets delivered to network layer because of sequence number
- The protocol looks like we have intermixed sender and receiver of the "simplex protocol in a noisy channel". *But it is not !!*
  - Piggy backing ACK on transmitted packets has side effects
  - Remember that this is a ***feedback*** system $\Rightarrow$ *small changes can cause significant results*
- If one side sends a packet and is received before the other side sends a packet, the protocol works perfectly
- If both sides send at the same time and the frames cross,
  - Half of the packets contain duplicates
  - Protocol is still correct, but bandwidth is wasted

A One-Bit Sliding Window Protocol (ctd.)

there are a lot of duplicates, and the bandwidth is wasted :(

bshr7 el abnormal fl note de.



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.

A One-Bit Sliding Window Protocol Disadvantages

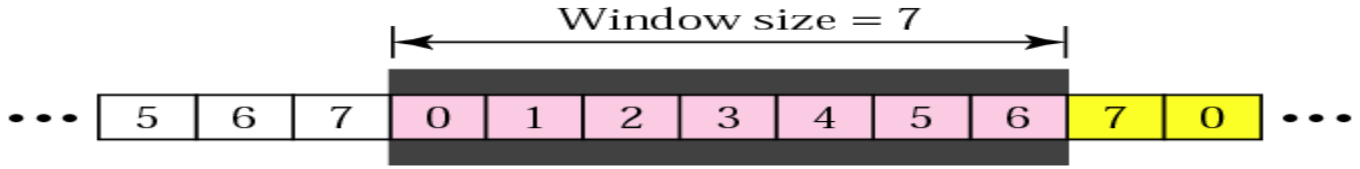
- Certain synchronization situation can result in continuous packet duplication
- Timeouts needs to be tuned carefully, otherwise multiple re-transmission can occur

da msh a7sn tare2a fl denya, laknha kant bdayet enhom y7sno 3leh 34an yewslo l 7aga a7sn kter, w enhom yst5dmo el pipelining wl piggypacking.

Go_Back_N ARQ

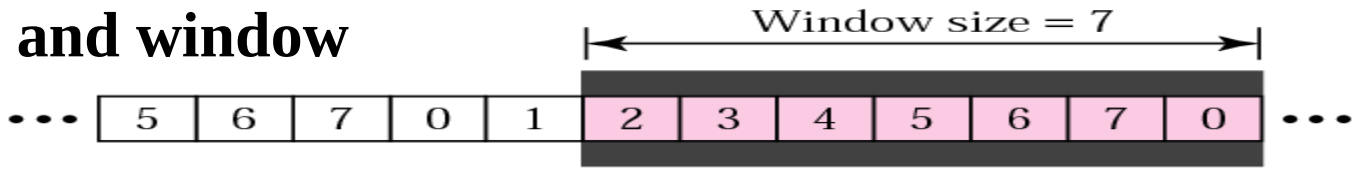
Sender sliding window

The sender window is an abstract concept defining an imaginary box of size $2^m - 1$ *(sequence numbers –1) The sender window can slide one or more slots when a valid acknowledgment arrives.*



a. Before sliding

If m = 3; sequence numbers = 8 and window size =7



b. After sliding two frames

Acknowledged frames

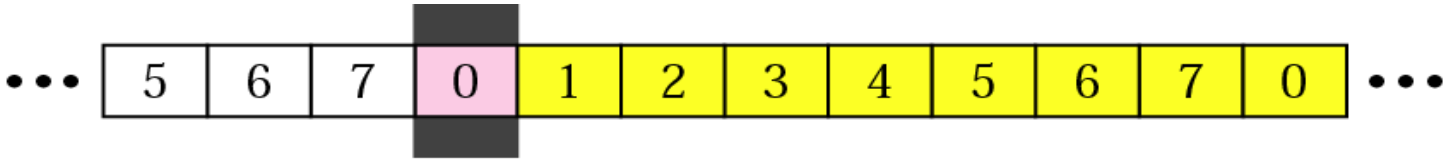
el window hena btt7rk 1 by 1, fa m3na eno ra7 lel 2, da m3nah en el reciever galo ack0, w ack1.

fe tre2a tanya lel sliding, en msh lazm ygele ack0, lw galy ack1, fa ana bb2a mzbt m3 el reciever, eno yb3tly ack lama yestlm 3dd mo3yn msln, fa hena lw oltelo yeb3tly kol ma yestlm 2 wra b3d, fa lw b3tly ack1 ana h3rf eno khlas estlm 0 w 1, w h7rk el window bt3ty khatwten odam.

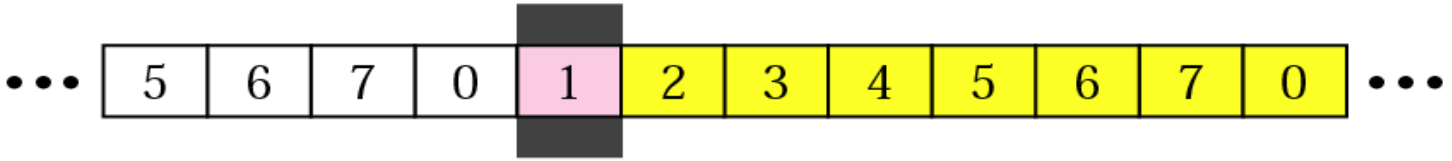
Go_Back_N ARQ

Receiver sliding window

- *The receive window is an abstract concept defining **an imaginary box of size 1** with one single variable Rn.*
- *The window slides when a correct frame has arrived; sliding occurs one slot at a time.*



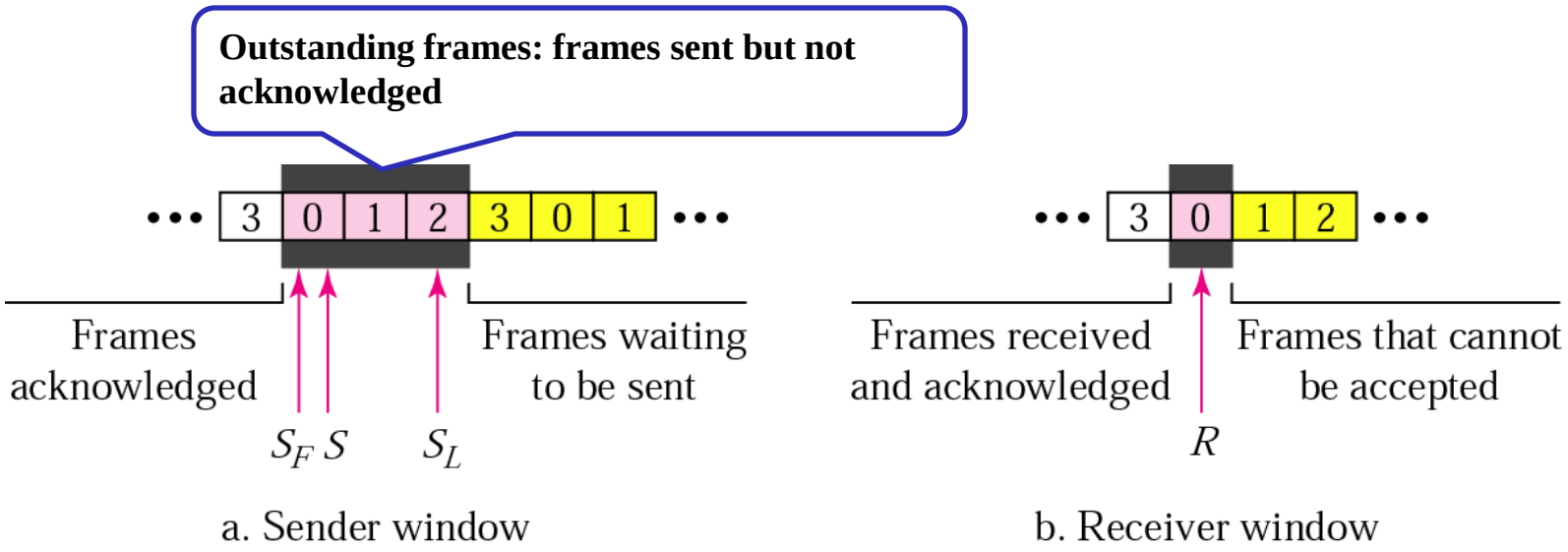
a. Before sliding



b. After sliding

Go-Back-N ARQ

control variables



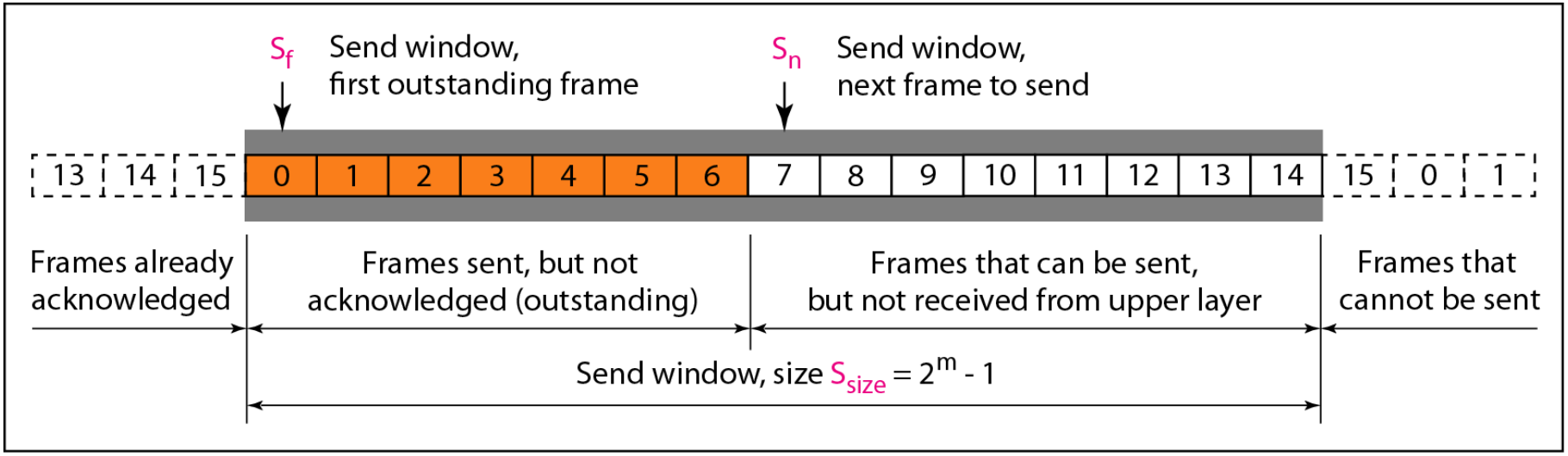
S: hold the sequence number of the recently sent frame

S_F: holds sequence number of the first frame in the window

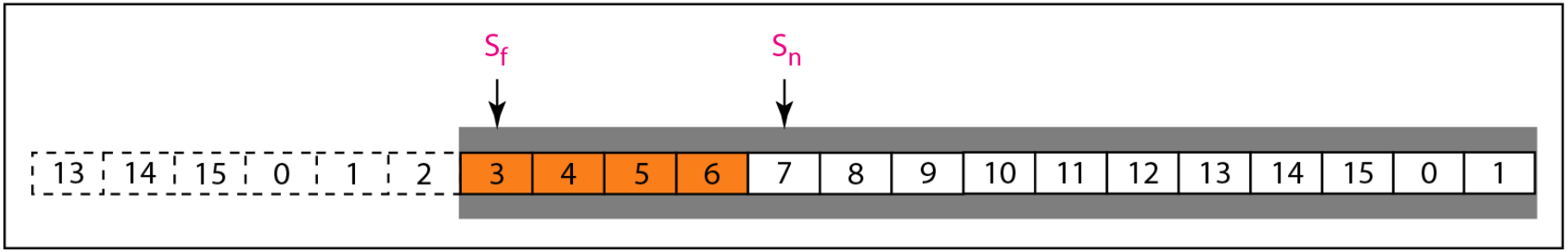
S_L: holds the sequence number of the last frame

R: sequence number of the frame expected to received

Go-Back-N ARQ



a. Send window before sliding



b. Send window after sliding

Go-Back-N ARQ

In Go-Back-N ARQ we use one timer for the first outstanding frame

- *The receiver sends a positive ACK if a frame has arrived safe and in order.*
- *if a frame is damaged or out of order ,**the receiver is silent** and will discard all subsequent frames*
- *When the timer of an unacknowledged frame at the sender site is expired , the sender goes back and resend all frames, beginning with the one with expired timer.(that is why the protocol is called Go-Back-N ARQ)*
- *The receiver doesn't have to acknowledge each frame received . It can send **cumulative Ack** for several frame*

Go-Back-N ARQ

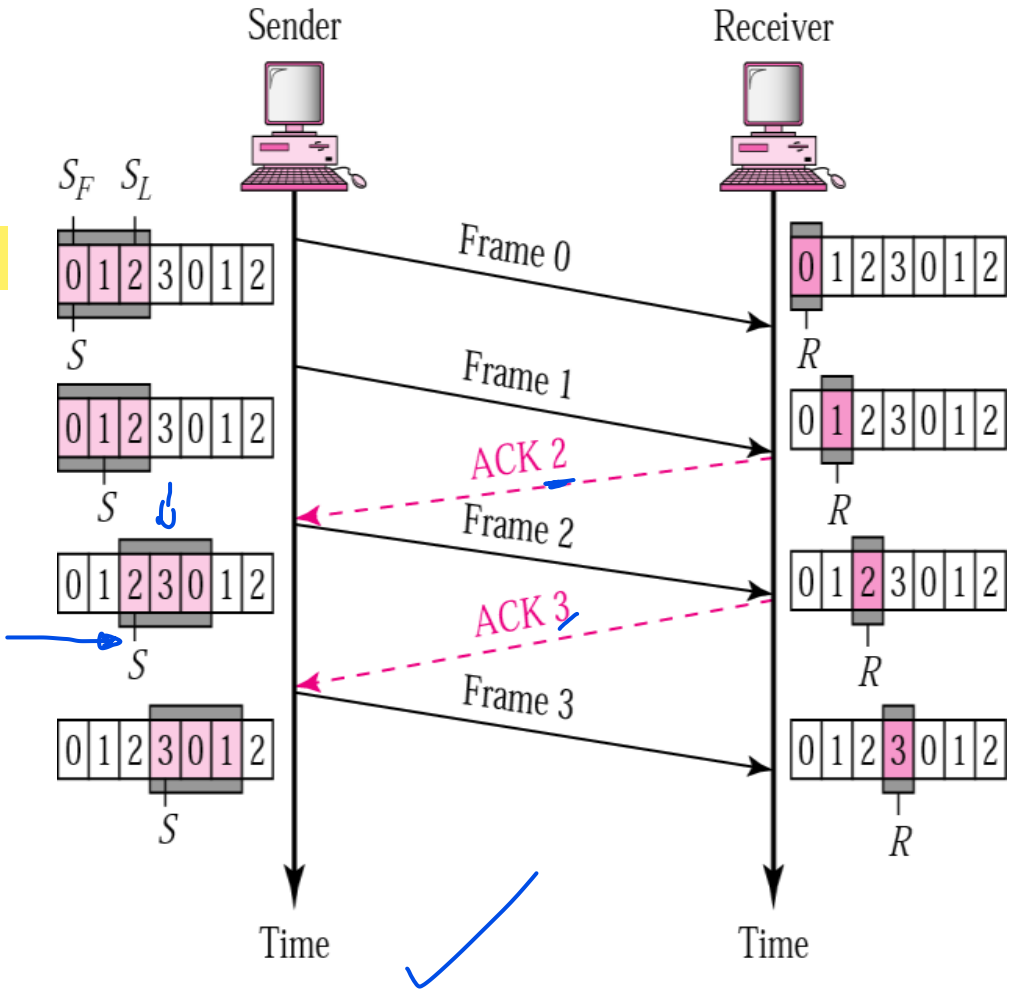
Example:

The sender has sent frame 6 , and timer expires for frame 3(frame 3 has not been acknowledge); the sender goes back and resends frames 3, 4,5 and 6

Go-Back-N ARQ

Normal operation

- *How many frames can be transmitted Without acknowledgment?*
- *ACK1 is not necessary if ACK2 is sent: Cumulative ACK*

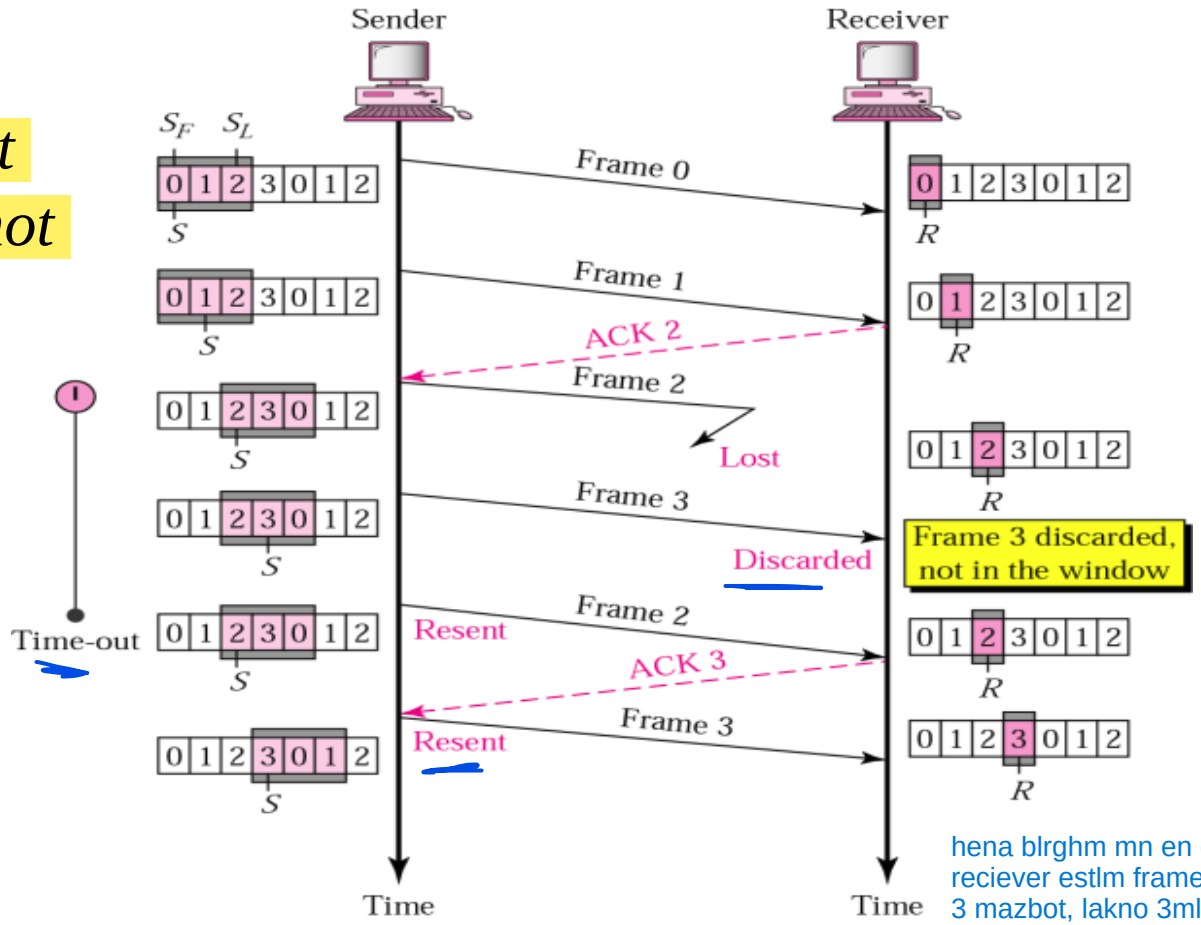


Go-Back-N ARQ

Damage or Lost Frame

Correctly received out of order packets are not Buffered

What is the disadvantage of this?

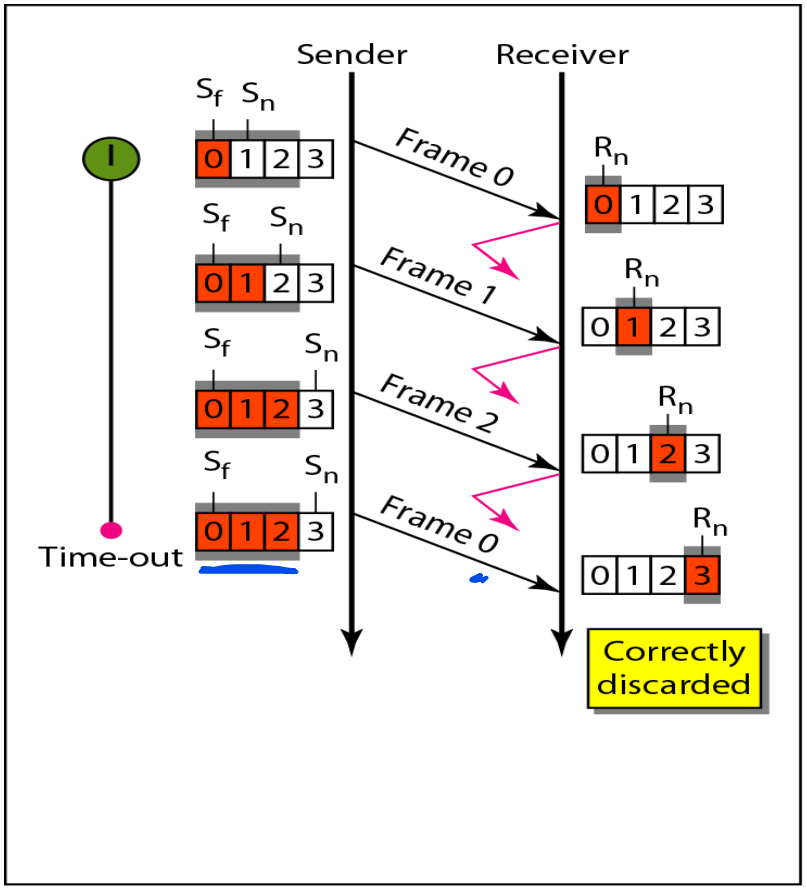


hena blrghm mn en el reciever estlm frame 3 mazbot, lakno 3mlo discard l2n howa kan mestny 2.

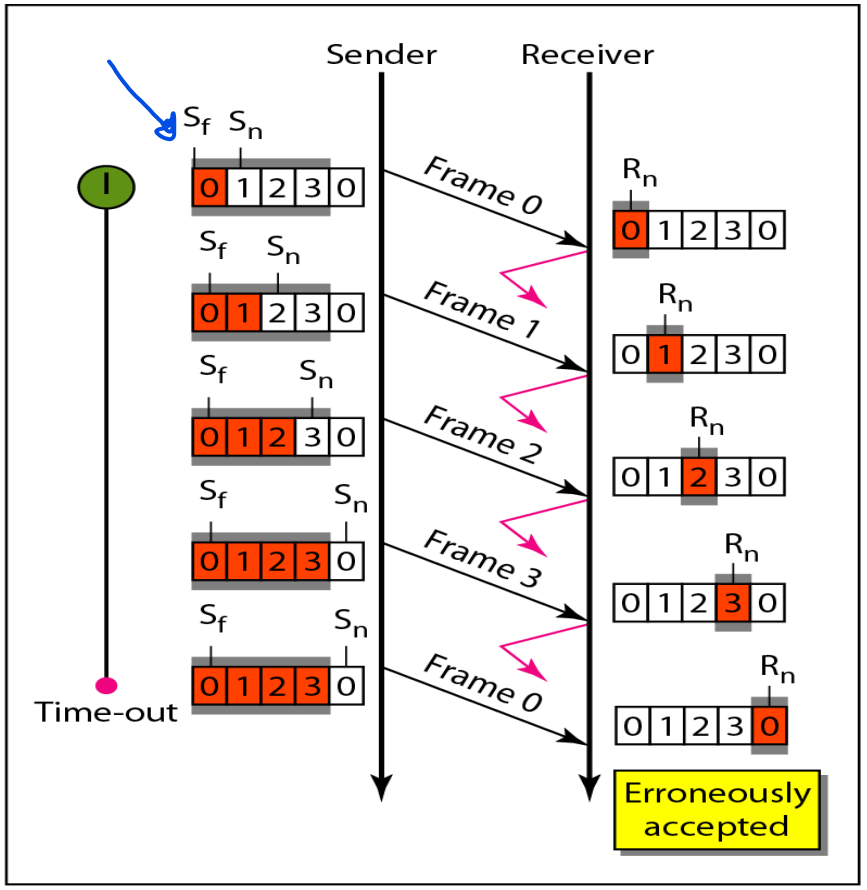
Go-Back-N ARQ

hena by3rfk hwa leh size of window is (2^m) - 1 msh 2^m.
el fekra en lw hwa 2^m, enta lw rkzt fl sora, emta el reciever est2bl 0, w el ack0 da3t, w est2bl 1, wl ack1 da3t .... l7d 3 w brdo el ack3 da3t.

delw2ty el mfrod en el sender htb3tlk nafs el packet el adema tany, l2n hwa 7sl 3ndo time out, fa lama hyegy yeb3t, hyb3tlk 0 bta3 el frame el adem.
lakn enta hena ht2rah akno frame geded l2n el reciever wa2f mstny 0.



a. Window size < 2^m



b. Window size = 2^m

lakn el kalam el fo2 da msh hy7sl fl case bta3t -1, l2n ana el 3nd el reciever hab2a mestny 3, w el time out ho2f 3nd 2, fa lama yegy yb3t tany hyb3tly 0, wana mstny delw2ty 3, fa el reciever msh hyt7rk khalas, w msh hy7sl moshkela.

Go-Back-N ARQ

In Go-Back-N ARQ, the size of the sender window must be less than $2^m = (2^m - 1)$ *; the size of the receiver window is always is 1..*

Bidirectional transmission : piggybacking

As Stop-and-Wait we can use piggybacking to improve the efficiency of bidirectional transmission . Each direction needs both a sender window and a receiver window.

Note

Stop-and-Wait ARQ is a special case of Go-Back-N ARQ in which the size of the send window is 1

Calculation of Window Size, Sequence number

- Total time = $T_t + 2$ Tp
- For 100% Efficiency → Window size = $(T_t + 2T_p)/ T_t$
- T_t : Transmission Time
- T_p : Propagation time

ehhna hena kol el m7tagen n3rfo, hat el Ws = 1 + 2Tp/Tt
w b3d keda hngeb el log bta3o w zy el fol keda.

- Ws = 1 + 2Tp/ Tt
- Log (Ws) = Log_2 (1 + 2 Tp/ T1) = Log_2 (1) + Log_2 (2 Tp / T1)
  - Log2(1)=0
- Example: Tt= 1 sec Tp= 49.5 sec
  - To get 100% efficiency : Ws = (1 + 2*49.5)/ 1 = 100
  - Min no. of bits in seq for 100% efficiency = log2 (100) = 7 bits
  - What if you use only no of bits is 6 , what is the max. efficiency you can get? You can use 2^6 seq numbers (64 seq numbers)
  - Window size = 64, so Efficiency is 64%

Go Back N algorithm

- Allow multiple outstanding frames at Sender
  - Outstanding frame: Frame sent but not yet acknowledged
- Sender cannot send more than MAX_SEQ to
  - enforce flow control
  - Avoid too much waste in case of packet loss/damage and large timeout
  - Certain incorrectness problems can occur if sender sends more than MAX_SEQ
- *Dropped* the assumption that network layer has *infinite* supply of packets
  - Network layer causes event ***network_layer_ready*** when it wants to send
- Need to enforce flow control of no more that MAX_SEQ outstanding buffers at sender
  - *enable_network_layer()* to allow network layer to send
  - *disable_network_layer()* to block network layer
- On timer expire, *all outstanding frames* are re-sent

Sliding Window Protocol Using Go Back N

```
/* Protocol 5 (pipelining) allows multiple outstanding frames. The sender may transmit up
   to MAX_SEQ frames without waiting for an ack. In addition, unlike 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                       /* should be 2^n – 1 */
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 */
}
```

- Window is between the sequence numbers ***a*** and ***c***
- ***a*** is considered earlier than ***c***
- Window is ***circular***
- Checks if ***b*** is within the window

- **Always** piggyback ACK with **every** data packet
- This means that one side may continue to get ACK even though it is not sending any traffic
- Hence, for this protocol, we **cannot** rely on **duplicate** ACK to infer that a packet is lost because the receiver may be sending reverse traffic at a very high rate and hence the reason the sender is receiving duplicate ACK is because the sent packets are still traveling on the way NOT that the sent packets were lost

- (Re-)Start a separate logical timer for every sent sequence number

- **s.ack** contains the sequence number of the last frame received
- Think of **s.ack** as ***circular(frame_expected -1)***
- *Remember that circular(x-1) = (x>0) ? (x--) : MAX_SEQ*

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;                   /* # 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 */
```

- Receiver window is of ***fixed size 1***
- Receiver window is between ***frame_expected*** and ***(frame_expected+1)***

Sender window is between ***ack_expected*** and ***next_frame_to_send***

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 */
         }
```

- We try to send packet in every loop
- If there is no other event, we send packets
- We disable network layer when sender window is full
- Hence we will always attempt to transmit the entire sender window

- Advance ***upper end*** of ***sender's*** window

⇒ increase sender's window

- Advance ***Lower end*** of ***receiver*** window
- *Remember* receiver window has **fixed size 1**

⇒ decrease 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 1 frame */
        inc(next_frame_to_send);  /* prepare to send the next one */
}
}

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

- Outstanding packets are between **ack_expected** and **next_frame_to_send**

e.g. if Ack 3
Then packets 2, 1,0,…, arrived
What is the benefit of this while loop?

- Advance ***lower end*** of Sender's window until it hits *r.ack*

⇒Reduce sender's window

⇒Free more buffers

- We have a separate timer per sent frame
- but when any timer expires, send all outstanding packets
- I.e. start sending from lower end of window (ack_expected)

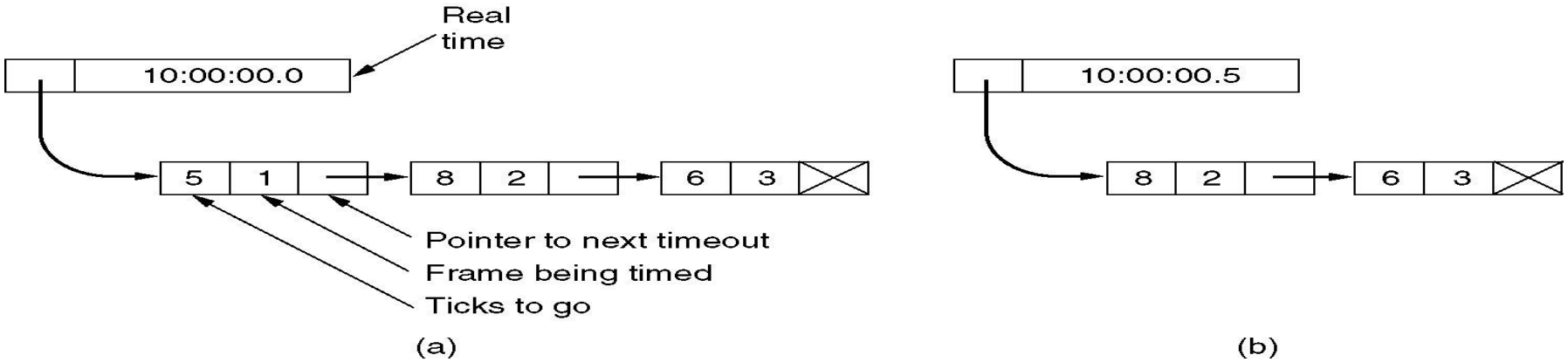
- Assume timeout because of frame loss
- Retransmit all frames ***starting*** from the ***last ACKed*** frame because receiver has window size one

- Max value of nbuffered is (MAX_SEQ) (*1 more* than maximum sequence number)
- Maximum number of outstanding packets is MAX_SEQ ***NOT*** (MAX_SEQ+1)
  - Because the network layer is enabled only of nbuffered is strictly less than MAX_SEQ
- I.e. there are at most MAX_SEQ+1 distinct sequence numbers

Sliding Window Protocol Using Go Back N Discussion points

- Circular sequence number is *MAX_SEQ*
- ACK is *piggy backed* with every packet on reverse traffic
  - If the sender is not transmitting packets, sender receives duplicate ACK (harmless)
  - If there is ***no reverse*** traffic, protocol ***fails***
    - sender will not receive ACK and not advance the window and hence will ***block***
    - Because sender did not receive ACK, it will keep timing-out and retransmitting
    - Suppose sender sends 1-2 packets and there is no more traffic at the sender
      - It seems like there is no failure here because all sent packets are received by the receiver
      - However there is still a failure (*Why is that considered failure?*)
- Sender window size (maximum outstanding frames) ***must be*** MAX_SEQ and ***not*** MAX_SEQ+1. I.e *one less than the maximum sequence* (*why*)
- Network layer is enabled only if `nbuffered` ***strictly less*** than MAX_SEQ
  - In the main loop, if there is no other event, the sender keeps sending up to the window size
  - Window size is MAX_SEQ ***not*** (MAX_SEQ+1)
- The receiver window size is 1 but we still need some buffering at the sender (*why*)
- We do ***not*** need buffering at the *receiver* (*why*)
- One ACK can be used for multiple buffers
  - ACK for frame *n* means that receiver received *(n-1)*, *(n-2)*,…, etc.
  - Good for lost or damaged ACKs
- Enable/disable network layer based on max outstanding buffers
- Need for *logically separate timer* per outstanding frame (*why*)
- Too many errors result in poor throughput because of retransmission

Sliding Window Protocol Using Go Back N Efficient Timer Implementation



- Sort timers by expiration time
- Put sorted timers in a queue (linked list)
- If timer T2 expires after timer T1, the "Ticks-to-go" field in timer T2 has the number of ticks that timer T2 expire after timer T1
- Every clock tick, decrement the timer at the head of the queue
- When the queue head expires, fire the timer and delete the timer from the queue
- Minimum processing per clock tick
- *Stop_timer* causes scan of the queue to find the timer that needs to be removed
- *Start_timer* causes scan of the queue to find where to insert the started timer and to update "Ticks-to-go" for timers ***after*** the inserted timers
- Argument of *start_timer* and *stop_timer* indicates which timer to start or stop

Selective Repeat ARQ

*Go-Back-N ARQ is inefficient in the case of a **noisy** link.*

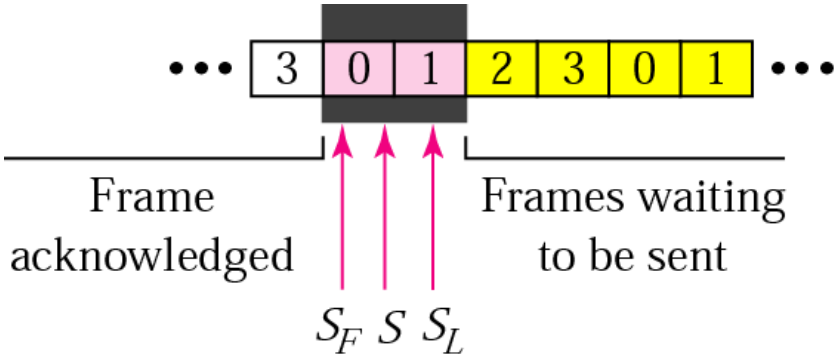
- *In a noisy link frames have higher probability of damage , which means the resending of multiple frames.*
- *this resending consumes the bandwidth and slow down the transmission .*

***Solution**:*

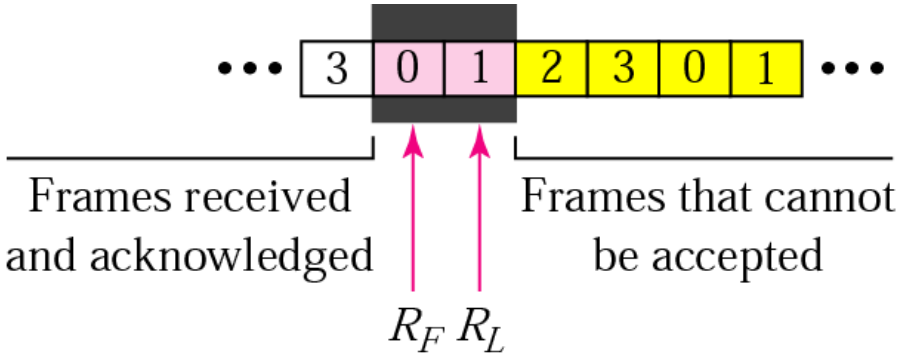
- *Selective Repeat ARQ protocol : resent only the damage frame*
- *It defines a **negative Acknolgment (NAK)** that report the sequence number of a damaged frame before the timer expires*
- *It is more efficient for noisy link, but the processing at the receiver is more complex*

Selective Repeat ARQ

- *The window size is reduced to one half of* 2^m
- *Sender window size = receiver window size =* 2^m */2*
- *Window size =* **(Max sequence number +1) /2**
- *If m = 2, Window size = 4/2=2*
- *Sequence number = 0, 1, 2 , 3*



a. Sender window

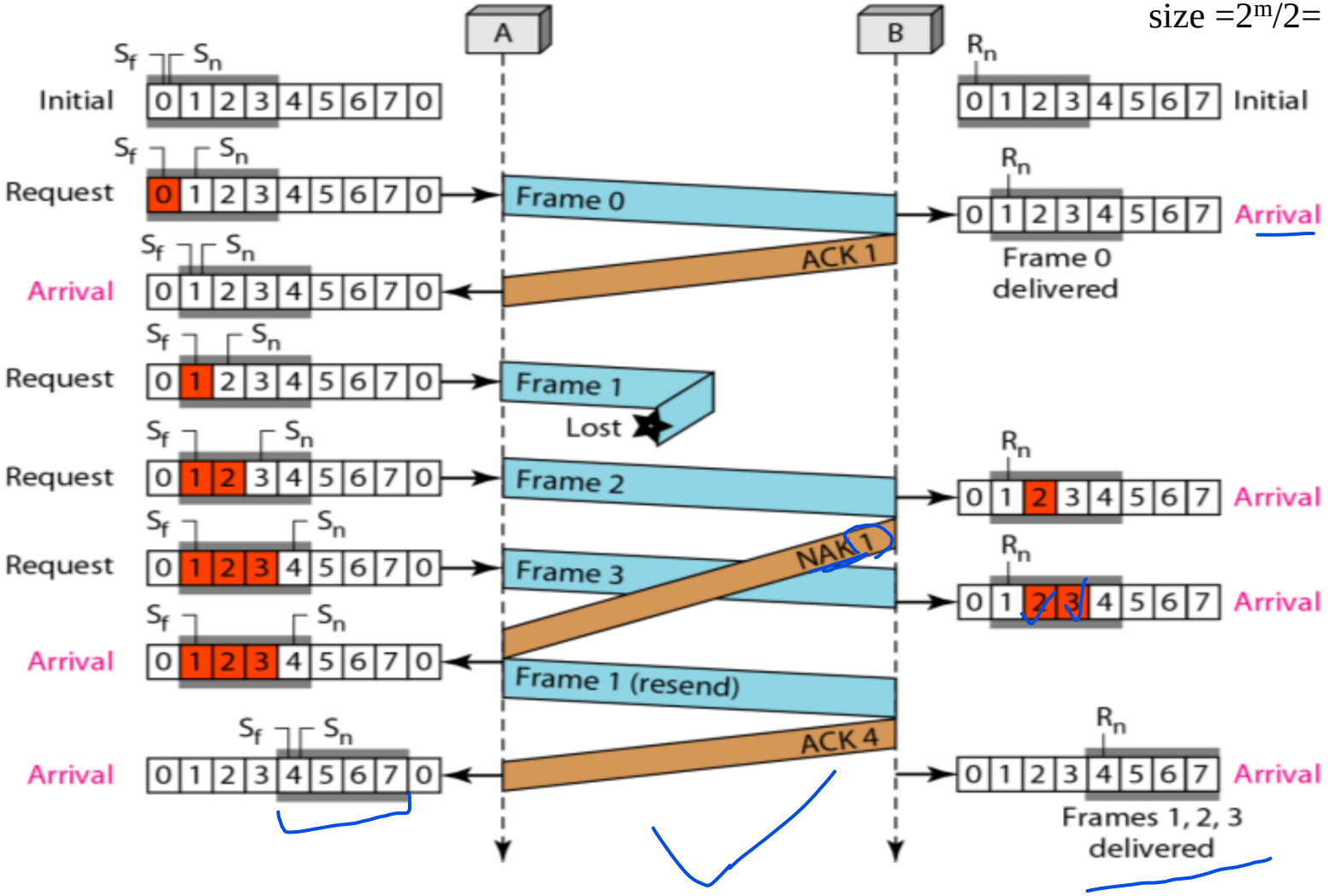


b. Receiver window

Selective Repeat ARQ

Lost Frame

m=3
Sequences no=2^m =8 :
0,1,2 ,3,4,5,6,7Window
size =2^m/2= 8/2=4



Selective Repeat ARQ

- *At the receiver site we need to distinguish between the acceptance of a frame and its delivery to the network layer.*
- *At the second arrival , frame 2 arrives and is stored and marked , but it can not be delivered because frame 1 is missing .*
- *At the next arrival , frame 3 arrives and is marked and stored , but still none of the frames can be delivered .*
- *Only at the last arrival , when finally a copy of frame 1 arrives , can frames 1 , 2 , and 3 be delivered to the network layer.*
- ***There are two conditions for the delivery of frames to the network layer: First , a set of consecutive frames must have arrived. Second, the set starts from the beginning of the window .***

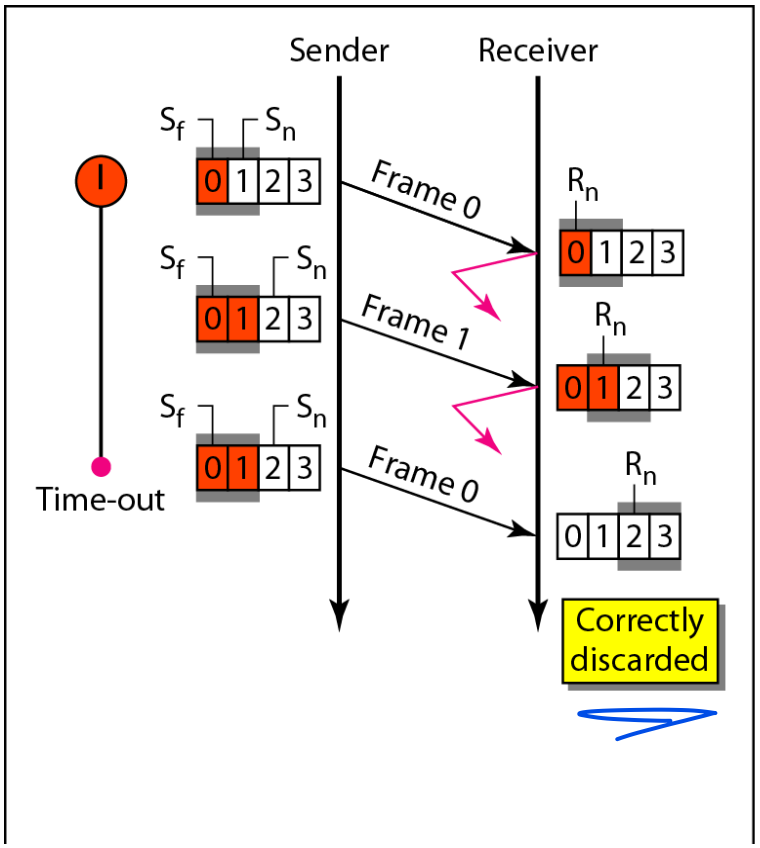
Selective Repeat ARQ

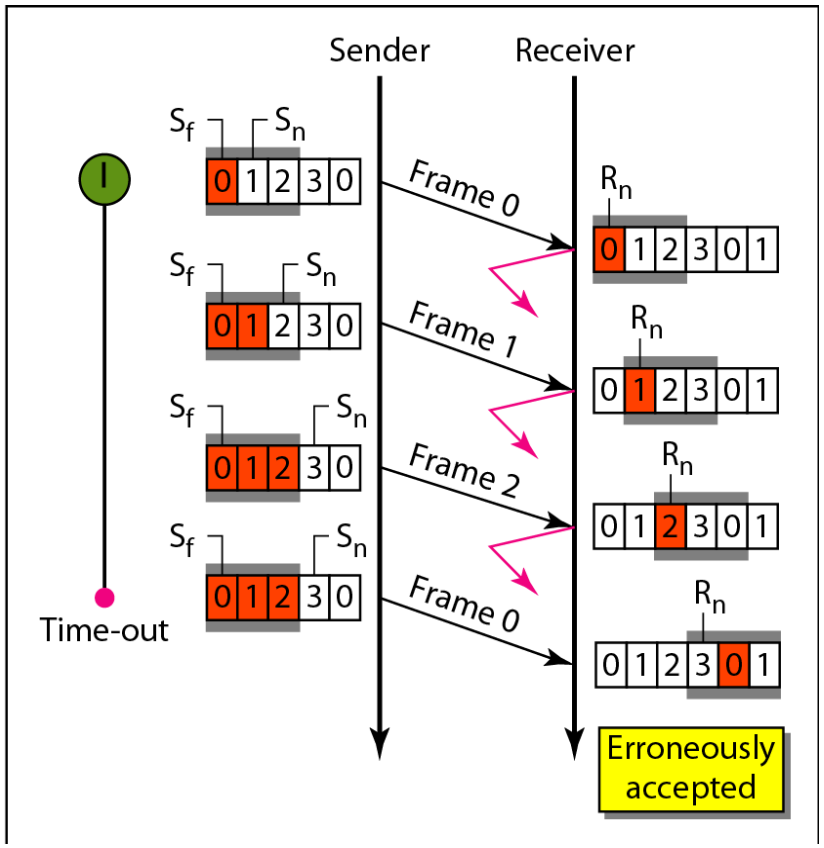
The next point is about the ACKs:

- *Notice that only two ACKs are sent here. The first one acknowledges only the first frame; the second one acknowledges three frames.*
- *In Selective Repeat, ACKs are sent when data are delivered to the network layer. If the data belonging to n frames are delivered in one shot , only one ACK is sent for all of them.*

Selective Repeat ARQ

m=2, Window size =2 , seq numbers is 4 (0,1,2,3)





why window size is 2^m / 2 nt 2^m

nfs fekret el 2^m -1 brdo, enk lw aktur mn keda,lama ye7sl error el reciever msh hy3rf, w hyftkr en dol packets gdeda msh el packets el adema.

bs hena baa brdo mynf34 2^m - 1, 34an hwa by5zn, fa hy5zn ghlt.

Note

In Selective Repeat ARQ, the size of In the sender and receiver window must be at most one-half of 2^m.

SR_ARQ Algorithms

Selective Repeat Algorithm

- Accepts ***out of order*** frames
- **One timer per frame**
- On timer expire, only the frame associated with timer is re-sent ⇒ *selective repeat*
- Receiver window size is **fixed** at (MAX_SEQ+1)/2
- Sender window size starts at 0 and *grows* to (MAX_SEQ+1)/2
- We have timer for ACK ⇒ No need for reverse traffic to piggyback ACK
- For each sequence number within window, receiver has
  - a buffer
  - "arrived" bit. If set, means buffer is full
- When a packet arrives at receiver
  - Check to see if it falls within window of sequence numbers
  - Check to see if arrive bit is clear (i.e. buffer is empty)
- If both conditions are satisfied, store the arrived packet
- Receiver delivers packets to network layer in order and advances window
- Assume that the function *start_ack_timer()* **resets** the expiration time of the ACK timer to the current time.

A Sliding Window Protocol Using Selective Repeat

Sender & Receiver window size is **half** of MAX_SEQ

```
/* Protocol 6 (nonsequential receive) 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 protocol5, 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 */
}
```

If frame type is ***not*** *data*, **s.seq** is ***ignored***
⇒ We can put anything

- ***s.ack*** contains the sequence number of the ***last received*** frame in the ***contiguous*** sequence of frames
- you can think of ***s.ack*** as **circular(frame_expected-1, MAX_SEQ)**

If the packet that we are sending now is a NAK, then clear **no_nak** This way if the next packet is NOT **frame_expected,** we will start the **ACK timer** instead of sending a NAK one more time

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;
```

Need buffer at ***both*** sender and receiver

Sender window is between ***ack_expected*** and ***next_frame_to_send***

receiver window is **fixed** size NR_BUFS. It is *initialized* between **0** and **NR_BUFs**

Continued →

A Sliding Window Protocol Using Selective Repeat (3)

out_buf[] is ***circular***

```
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 */
                   }
              }
         }
```

- We did NOT receive the lower end of receive window
⇒Send NAK immediately
- Because ***send_frame()*** sets ***s.ack*** to **circular(frame_expected-1)** then the NAK frame contains the sequence number of the last received frame
- Thus a when a sender receives a NAK, it should re-send the frame with sequence number **circular(r.ack+1)**

Because we are sending a NAK, this value is meaningless ⇒ Put any thing

Restart ACK timer every time we receive out of order packet (***See WHY later?***)

Deliver ***contiguous*** packet sequence to network layer starting from **bottom** of receiver window. ***frame_expected*** is ***one more than*** the sequence number of the **last received frame** in ***contiguous*** sequence

- Increment **both** lower and upper receiver window boundaries
⇒ Receiver window size *always* fixed at **NR_BUFs**

Allow sending NAK because we send NAK for ***frame_expected*** and ***frame_expected*** will be incremented soon

Reset ACK timer to the current time

A Sliding Window Protocol Using Selective Repeat (4)

- When the other side sends a NAK, the other side sets the **"ack"** field to **circular (frame_expected – 1).**
- Hence **r.ack** contains sequence number of the last received frame by the other side.
- Thus a sender should re-send the frame whose sequence number is **circular(r.ack+1)**
- Thus we have to test ***between*** for **circular(r.ack+1)**

- ACK means all frames in the **contiguous** sequence of frames **before and including** ACKed frame have been received. Do the following *between* **ack_expected** and **r.ack**

⇒Free buffers
⇒Stop all retransmit timers
⇒Advance sender **lower** window

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

We assume that the timeout event causes the variable ***oldest_frame*** to be set according to the timeout that just expired

Re-send NAK for last frame received if we get a corrupted frame

- On ACK timeout, we send ACK for the frame **before** bottom of the receiver window because **frame_expected** is one more than the sequence number of the last frame received in **contiguous** sequence
- Because *send_frame()* sets ***s.ack*** to **circular(frame_exptected-1)**, we will end up *re-ACKing* the last frame received in the **contiguous** sequence

Because we have ACK timeout
⇒No need for reverse traffic
⇒Solves the blocking problem of Go back N

Window size is NR_BUF
⇒ half of MAX_SEQ

A Sliding Window Protocol Using Selective Repeat Discussion Points

- Why is NR_BUFS defined (MAX_SEQ+1)/2?
- How many buffers must receiver have?
- How may timers are needed at the receiver?
- How to get away without reverse traffic?
- How is NAK making protocol more efficient
- How to adjust timers
- ***Why do we have to restart ACK timer every time we receive out of order packet?***

A Sliding Window Protocol Using Selective Repeat Why NR_BUFS (MAX_SEQ+1)/2

- Suppose we allow NR_BUFS to be MAX_SEQ-1 (just like Go Back N)
- Consider the following sequence of events
- Sender window is 0,1,2,3,4,5,6
  - `next_frame_to_send` = 7, `ack_expected` = 0
- Receiver window is 0,1,2,3,4,5,6
- Sender sends frames 0,1,2,3,4,5,6
- Receiver receives all seven frames and sends ACK for all of them
  - All frames are valid frames because their sequence numbers lie within receiver window
  - Receiver delivers all 7 frames to network layer and sends ACK for 0,1,2,3,4,5,6
  - Receiver advances its receive window to 7,0,1,2,3,4,5
    - $\Rightarrow$ `Frame_expected` = 7, `too_far` = 6
- Suppose all ACKs are lost
- Sender times out. Thus it **resends** frame 0 because frame 0 timer is the first timer started and hence it will be the first timer to expire
- Receiver receives **resent** data frame with s.seq = 0
  - S.ack = 0 is within the new receiver window
  - Receiver accepts frame 0 $\Rightarrow$ **DUPLICATE** packet because this packet is already delivered to network layer
  - arrived[0] = TRUE
  - Does NOT advance `frame_expected` because `frame_expected` not received yet

A Sliding Window Protocol Using Selective Repeat Why NR_BUFS (MAX_SEQ+1)/2

- Now receiver ACK timeout expires
  - Resends ACK with s.ack = circular(`frame_expected` – 1) = 6
- Sender receives ACK with s.ack = 6
  - Sender assumes that all frames 0,1,2,3,4,5,6 have been received
  - Sender advances its window to so that the sender window = 7,0,1,2,3,4,5
  - Sender sends frames 7,0,1,2,3,4,5

  ⇒ `ack_expected` = 7, `next_frame_to_send` = 6
- Receiver receives frame 7,0,1,2,3,4,5
  - Receiver **accepts** frame 7 and puts it in buffer
  - Receiver **rejects** frame 0 because arrived[0]=TRUE ⇒ The new frame 0 is **LOST**
  - Receiver **Accepts** 1,2,3,4,5 and puts them in buffer
  - Now receiver has a contiguous sequence of frames in the receive window so it delivers 7,0,1,2,3,4,5 to network layer
  - The delivered packets have the following problems
    - Frame 0 is **duplicate** and **out of sequence** because it belongs to the first batch
    - Frame 0 from the second batch is **lost**
- NOTE:
  - When Receiver receives frame 0 for the second time, it will send a NAK for frame_expected. Thus s.ack=6 in the NAK
  - The sender receives a NAK with r.ack = 6
  - The sender checks if **circular(r.ack+1)** lies between **ack_expected** and **next_frame_to_send** before resending the NAKed frame
    - circular(r.ack+1) = 7, ack_expected = 0, and next_frame_to_send = 7
    - Thus the check **Fails**
    - Hence the NAK will have **no effect**

A Sliding Window Protocol Using Selective Repeat How many buffers and timers at receiver

- Receiver only accepts frames within the window size
- Window size is **half** the sequence number*
- Receiver needs buffers equal to window size **not** MAX_SEQ
- We need one timer ONLY at receiver: This is the ACK timer
- We need timers equal to window size, not MAX_SEQ at the sender

A Sliding Window Protocol Using Selective Repeat Auxiliary Timer and NAK

- Auxiliary ACK timer
- No need for reverse traffic
- Auxiliary ACK timer should be *shorter* than retransmit timer
- NAK to expedite retransmit (not needed for correctness)
- Send NAK on
  - Out of sequence: `r.seq ≠ frame_expected`
  - Checksum error: possible damage
  - In both cases, we send NAK for **circular(frame_expected – 1)**
  - Remember that *circular(x-1) = (x>0) ? (x--) : MAX_SEQ*
- How to avoid multiple NAKs per packet
  - Variable `no_nak` is set only if receiver has **not** sent NAK for `frame_expected`

A Sliding Window Protocol Using Selective Repeat Adjusting timers

- Sender timeout at sender should be *slightly larger* than roundtrip time
- Problem if round trip time is variable
- If reverse traffic is sporadic, ACK will be irregular, making it hard for receiver to estimate roundtrip time

- Receiver auxiliary ACK timer should be appreciably **shorter** than sender retransmit timeout

Why Restart ACK timer For Out of Order packet?

- Consider the following sequence of events
  - Sender sends packets 0,1,2,3
  - Receiver receives packets 0,1,2,3
  - Receiver sends ACK for all of them and advances window to 4,5,6,7
  - All ACKs are lost
  - *How will the protocol recover?*

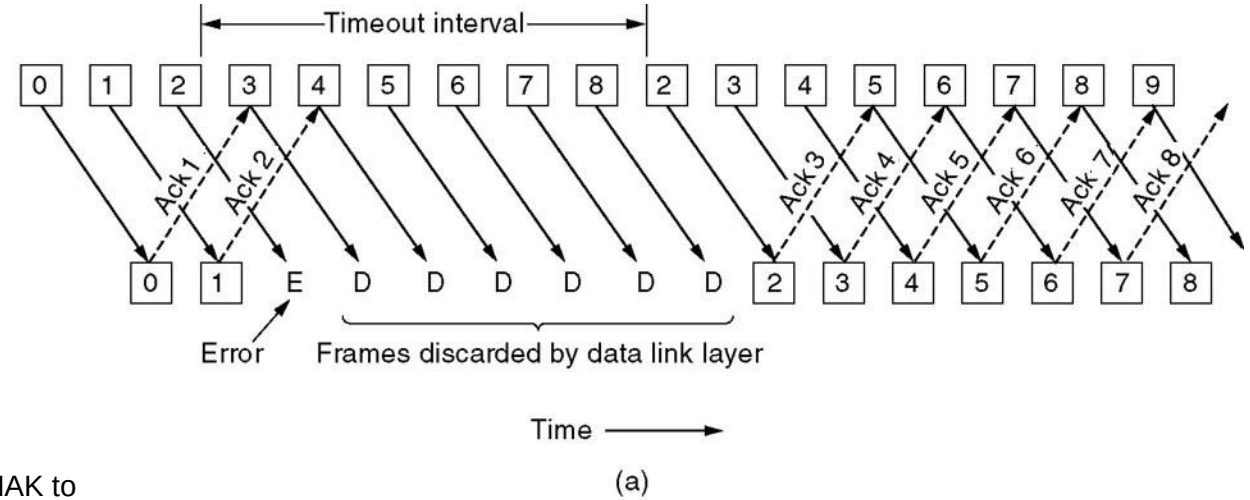
Go Back N vs. Selective Repeat

- Go Back N
  - Receiver discards all frames after lost or damaged frame
  - Receiver acknowledges received frame
  - Receiver does NOT send any ACK for frames received after lost or damaged frame
  - Relies on sender timeout
  - Equivalent to receiver window of size **1**
  - Wastes a lot of bandwidth in case of error
  - Receiver needs to *buffer 1 frame* only
- Selective Repeat
  - Receiver buffers all frames received within window
  - Receiver acknowledges *last received in sequence frame* for every out of sequence frame
    ⇒ **Cumulative** ACK
  - On timeout, sender only re-transmits *oldest unacknowledged* frame
  - Receiver can deliver all buffered frames, *in sequence*, to the network layer
  - Receiver often use NAK* to stimulate retransmission before timeout
  - Receiver needs to *buffer multiple frames* up to window size
- *Tradeoff between buffer space and link bandwidth utilization*

Go Back N vs. Selective Repeat

Expected frame not last received frame

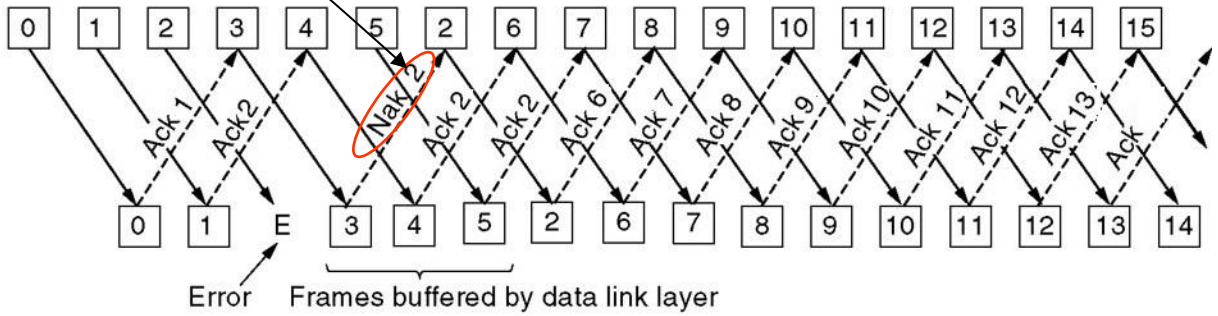
Go Back N



(a)

Use NAK to stimulate retransmit

Selective Repeat with NAK



(b)

Example Data Link Protocols

- HDLC – High-Level Data Link Control
- PPP - The Data Link Layer in the Internet

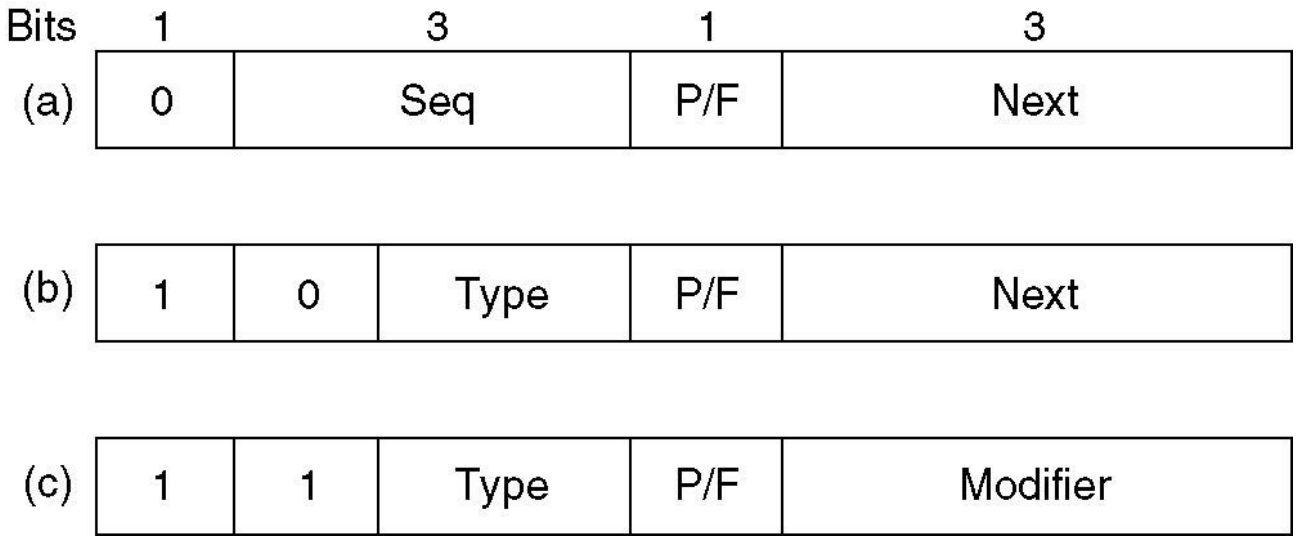
High-Level Data Link Control

- Frame format for bit-oriented protocols.
- HDLC uses bit stuffing and allows non-integer frames of, say, 30.25 bytes
- HDLC provides reliable transmission with a sliding window

Bits	8	8	8	≥ 0	16	8
	0 1 1 1 1 1 1 0	Address	Control	Data	Checksum	0 1 1 1 1 1 1 0

High-Level Data Link Control (2)

P/F Poll/Final bit



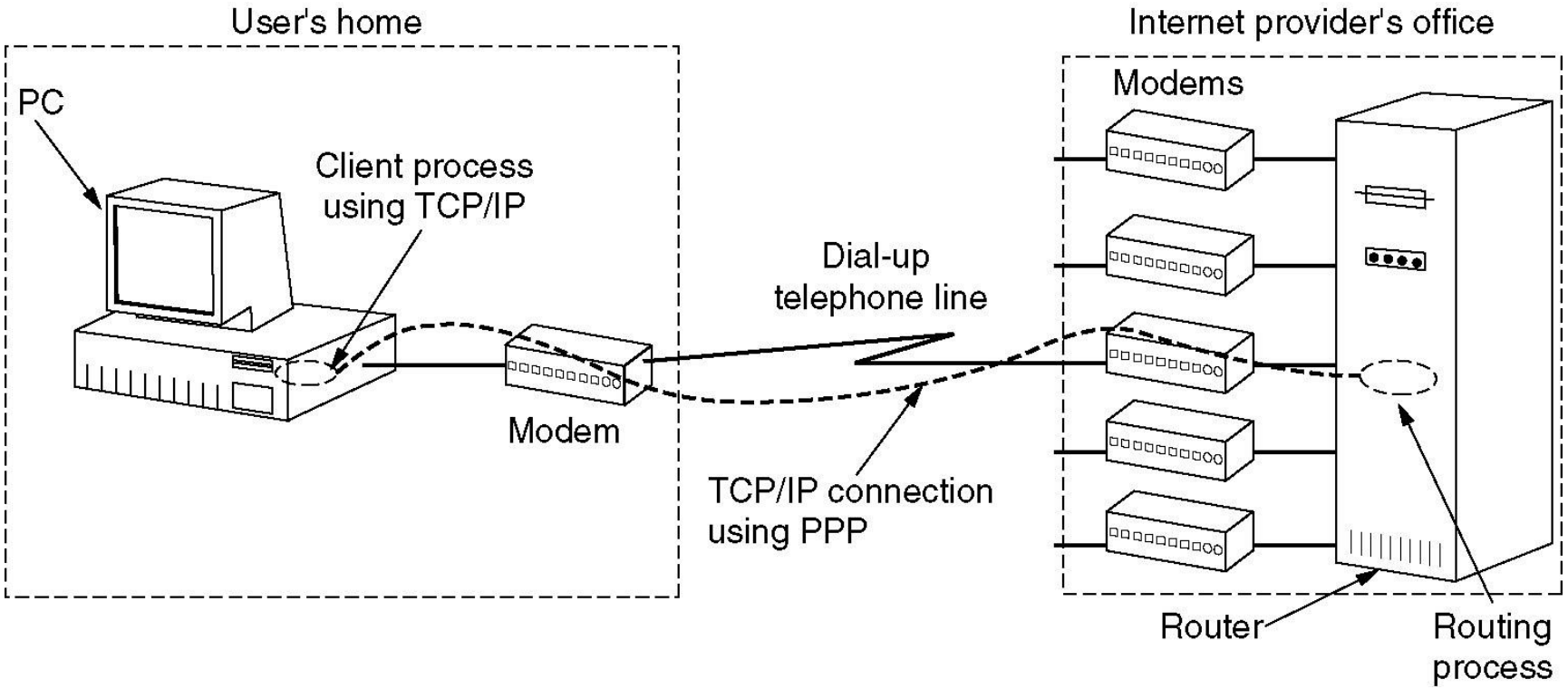
(a) An information frame starts with 0.

(b) A supervisory frame 10: control functions such as acknowledgment of frames, requests for re-transmission, and requests for temporary suspension of frames being transmitted

(c) An unnumbered frame 11: also used for control purposes. It is used to perform link initialization, link disconnection and other link control functions.

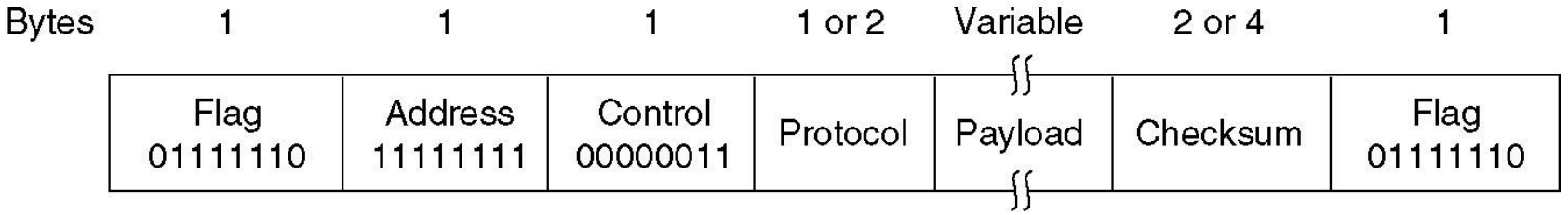
The Data Link Layer in the Internet

A home personal computer acting as an internet host.



PPP – Point to Point Protocol

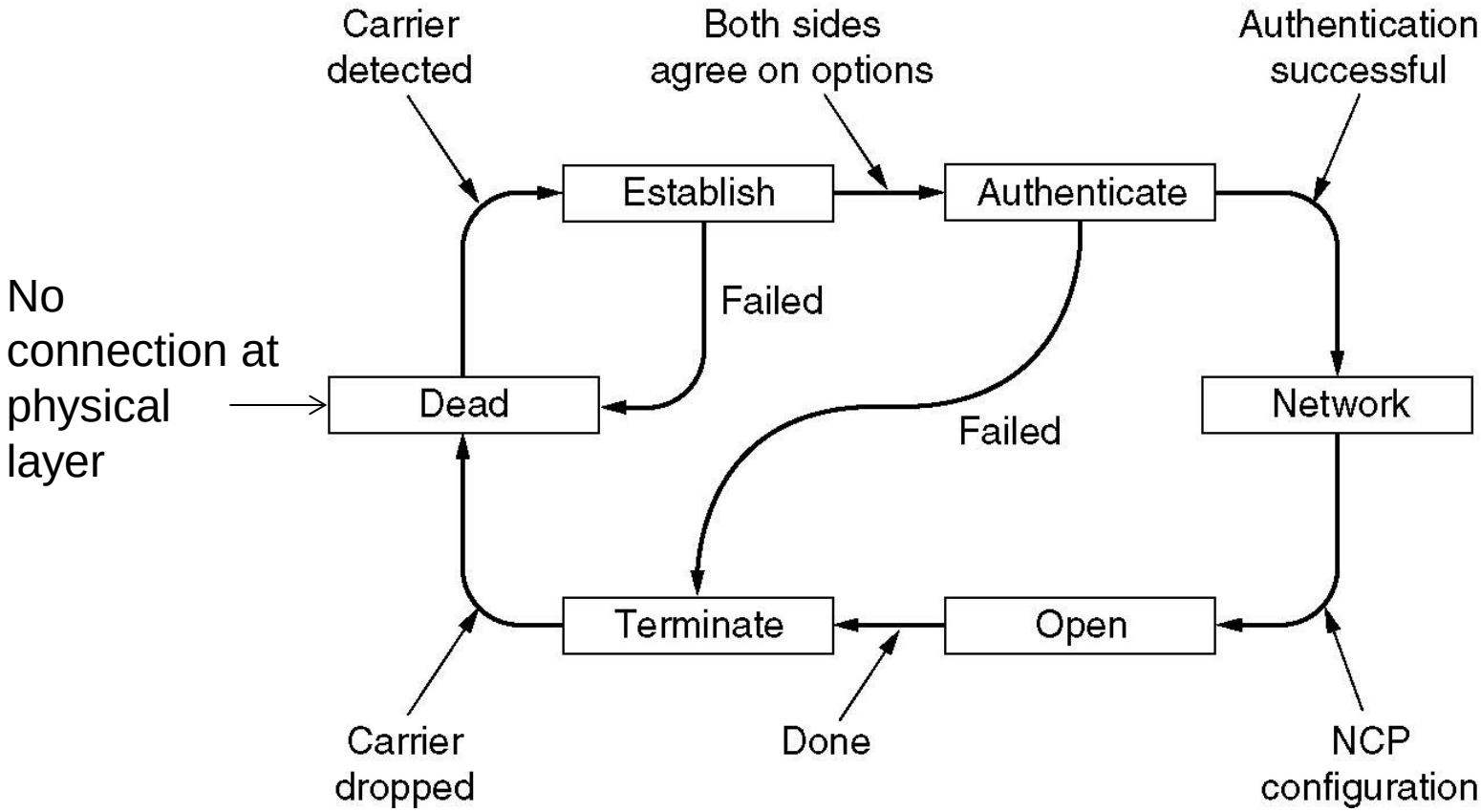
- The PPP full frame format for unnumbered mode operation.
- PPP uses byte stuffing and all frames are an integral number of bytes



Address field: This field is always set to the binary value 11111111 to indicate that all stations are to accept the frame
Control field: the default value of which is 00000011. This value indicates an unnumbered frame
Protocol field: To tell what kind of packet is in the Payload field

- Codes starting with 0 for information
- Codes starting with 1 are used for configurations (Link Control Protocol LCP, Network control Protocol NCP)

PPP – Point to Point Protocol (2)



A simplified phase diagram for bring a line up and down.

PPP – Point to Point Protocol (3)

The LCP frame types.

Name	Direction	Description
Configure-request	I → R	List of proposed options and values
Configure-ack	I ← R	All options are accepted
Configure-nak	I ← R	Some options are not accepted
Configure-reject	I ← R	Some options are not negotiable
Terminate-request	I → R	Request to shut the line down
Terminate-ack	I ← R	OK, line shut down
Code-reject	I ← R	Unknown request received
Protocol-reject	I ← R	Unknown protocol requested
Echo-request	I → R	Please send this frame back
Echo-reply	I ← R	Here is the frame back
Discard-request	I → R	Just discard this frame (for testing)