CSCI 460
Networks and Communications

Datalink Layer

Humayun Kabir
Professor, CS, Vancouver Island University, BC, Canada

Outline

- Connectionless services
- Framing
- Error Control
- Error-Correcting Code
- Error-Detecting Code
- Flow Control
  - Stop and Wait Protocol
  - Sliding Window Protocol with Go Back N
  - Sliding Window Protocol with Selective Repeat

Connectionless Services

Unacknowledged connectionless service

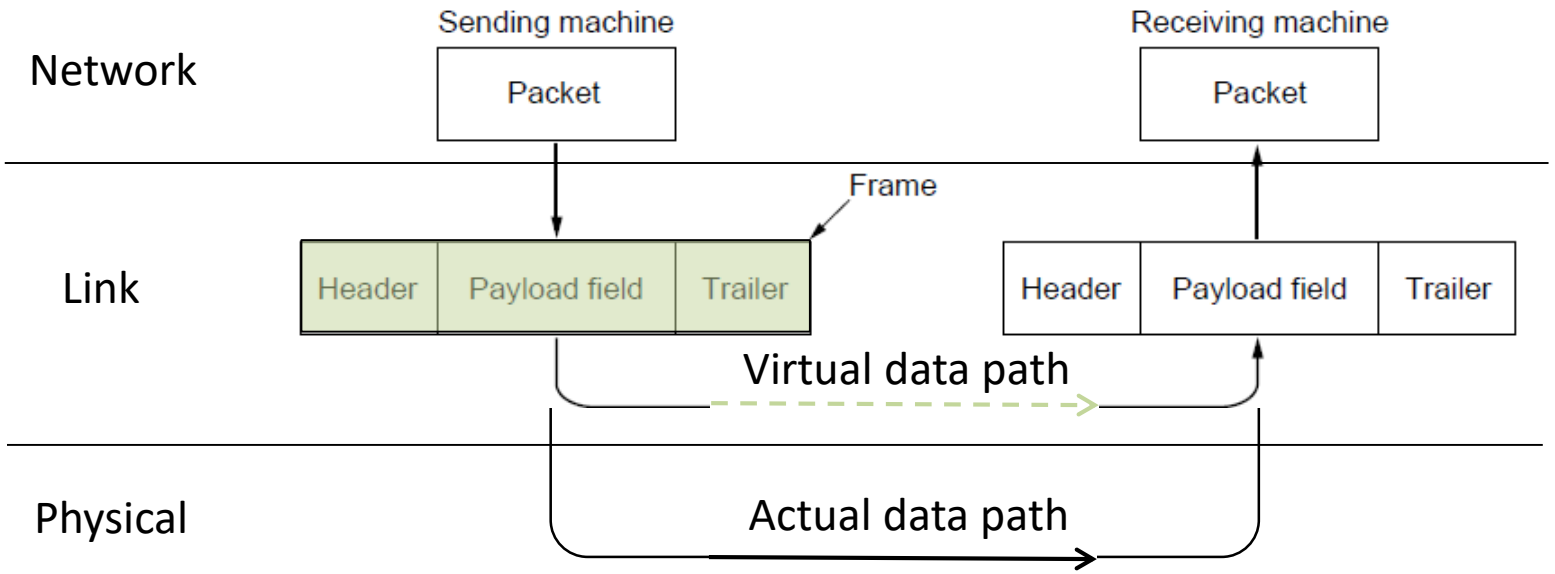
- Frame is sent with no connection and error recovery
- Ethernet is example

Acknowledged connectionless service

- Frame is sent with no connection but with retransmissions if needed
- Example is 802.11

Frames

Link layer accepts packets from the network layer, and encapsulates them into frames that it sends using the physical layer; reception is the opposite process



Framing Methods

- Frames need to be delimited from each other by the sender before transmitting over the physical medium in order to facilitate the receiver to separate them upon receptions.
- Once the frames are separated from each other these delimiters have no other usage at the receiver.
- Frame delimiters are not part of data link layer protocol header.

Framing Methods

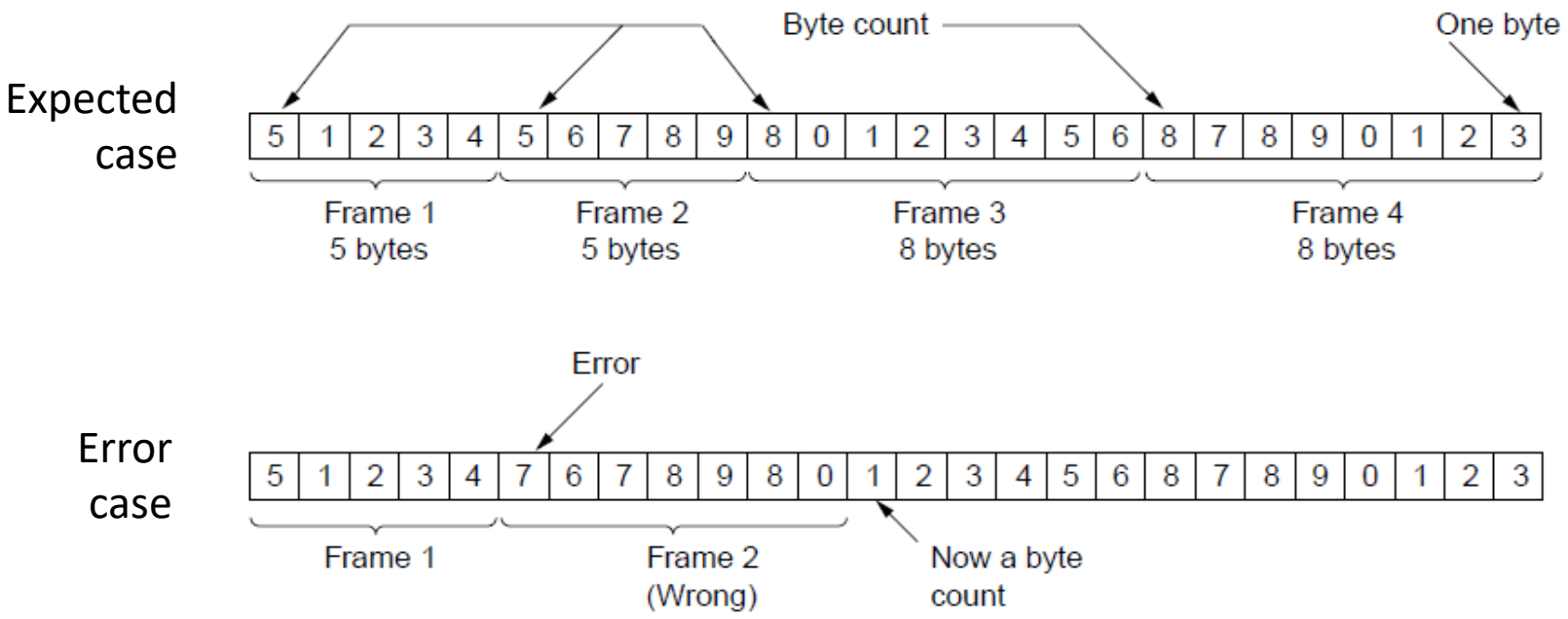
Frame delimiting methods

- Byte count »
- Flag bytes with byte stuffing »
- Flag bits with bit stuffing »
- Physical layer coding violations
  - Use non-data symbol to indicate frame

Framing – Byte count

Frame begins with a count of the number of bytes in it

- Simple, but difficult to resynchronize after an error

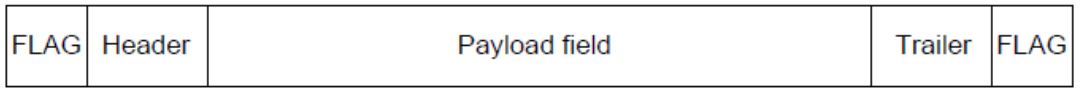


Framing – Byte stuffing

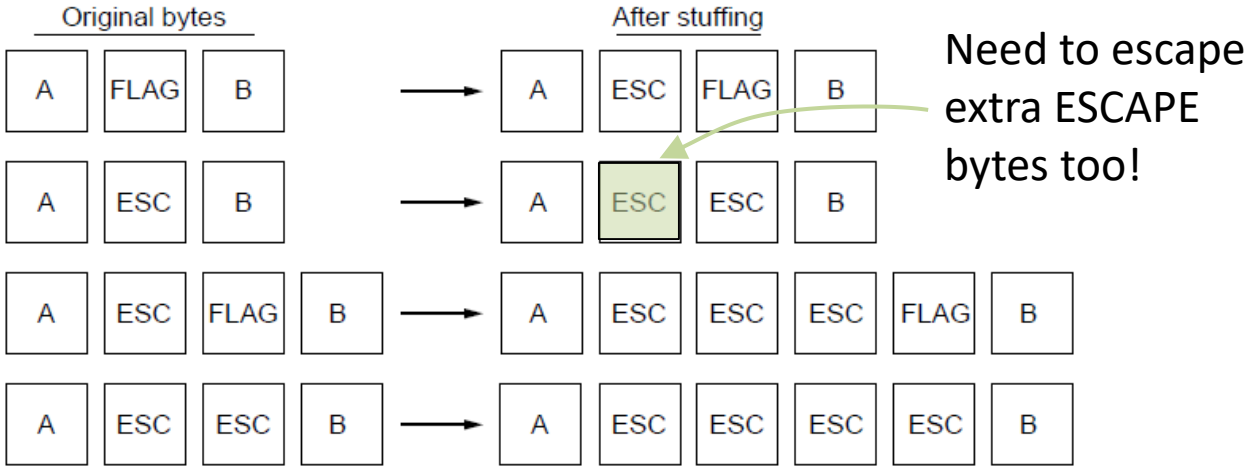
Special flag bytes delimit frames; occurrences of flags in the data must be stuffed (escaped)

- Longer, but easy to resynchronize after error

Frame format



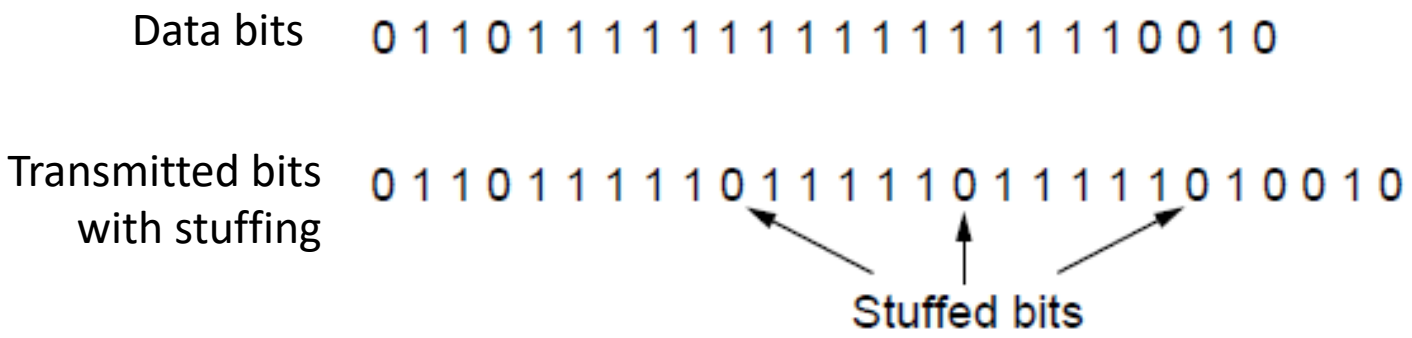
Stuffing examples



Framing – Bit stuffing

Stuffing done at the bit level:

- Frame flag has six consecutive 1s (not shown)
- On transmit, after five 1s in the data, a 0 is added
- On receive, a 0 after five 1s is deleted



Error Control

Error control repairs frames that are received in error

- Requires errors to be detected at the receiver
- Requires to acknowledge error free frames
- Typically retransmits the unacknowledged frames
- Timer protects against lost acknowledgements

Error control codes add structured redundancy to data so errors can be either detected, or corrected.

Error Detection and Correction

Error correction codes:

- Hamming codes »
- Binary convolutional codes »
- Reed-Solomon and Low-Density Parity Check codes
  - Mathematically complex, widely used in real systems

Error detection codes:

- Parity »
- Checksums »
- Cyclic redundancy codes »

Error Bounds – Hamming distance

Code turns data of ***m*** bits into some codewords of (***m***+***r)*** bits

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

- Example with 4 codewords of 10 bits (***m***=2, ***r***=8):
  - 0000**00**0000, 0000**01**1111, 1111**10**0000, and 1111**11**1111
  - Hamming distance is 5

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)

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 with 7 bits message and can correct 1 bit error either in a check bit or in a message bit

Error Correction – Hamming code

7-bit message to send 1 0 0 0 0 0 1

- Check bits are placed at the **power** of **twos positions**; e.g. 1, 2, 4, and 8 in the **codeword**.
- Message bits are placed at the **rest** of the **positions**; e.g. 3, 5, 6 ,7,9, 10, and 11 in the **codeword**..
- Codeword with check bits and message bits: $P_1P_2M_3P_4M_5M_6M_7P_8M_9M_{10}M_{11}$
- Check bits are **parity function** over the subsets of the message bits.

Error Correction – Hamming code

Message to send 1 0 0 0 0 0 1

- Message bit positions can be expressed as the sum of the positions of the check bits with the least number of terms in the sum.

P_1	P_2	M_3	P_4	M_5	M_6	M_7	P_8	M_9	M_{10}	M_{11}
P_1	P_2	1	P_4	0	0	0	P_8	0	0	1

	P_1	P_2	P_4	P_8
3	= 1 + 2			
5	= 1 + 0 + 4			
6	= 0 + 2 + 4			
7	= 1 + 2 + 4			
9	= 1 + 0 + 0 + 8			
10	= 0 + 2 + 0 + 8			
11	= 1 + 2 + 0 + 8			

$P_1 = \mathbf{f}(M_3, M_5, M_7, M_9, M_{11}) = f(1,0,0,0,1) = 0$

$P_2 = \mathbf{f}(M_3, M_6, M_7, M_{10}, M_{11}) = f(1,0,0,0,1) = 0$

$P_4 = \mathbf{f}(M_5, M_6, M_7) = f(0,0,0) = 0$

$P_8 = \mathbf{f}(M_9, M_{10}, M_{11}) = f(0,0,1) = 1$

Codeword sent

P_1	P_2	M_3	P_4	M_5	M_6	M_7	P_8	M_9	M_{10}	M_{11}
0	0	1	0	0	0	0	1	0	0	1

Error Correction – Hamming code

Code word received

P_1	P_2	M_3	P_4	M_5	M_6	M_7	P_8	M_9	M_{10}	M_{11}
0	0	1	0	1	0	0	1	0	0	1

Re-compute check bits

P_1	P_2	M_3	P_4	M_5	M_6	M_7	P_8	M_9	M_{10}	M_{11}
P_1	P_2	1	P_4	1	0	0	P_8	0	0	1

	P_1	P_2	P_4	P_8
3	= 1	+ 2		
5	= 1	+ 0	+ 4	
6	= 0	+ 2	+ 4	
7	= 1	+ 2	+ 4	
9	= 1	+ 0	+ 0	+ 8
10	= 0	+ 2	+ 0	+ 8
11	= 1	+ 2	+ 0	+ 8

$P_1 = \mathbf{f}(M_3, M_5, M_7, M_9, M_{11}) = f(1,1,0,0,1) = 1$

$P_2 = \mathbf{f}(M_3, M_6, M_7, M_{10}, M_{11}) = f(1,0,0,0,1) = 0$

$P_4 = \mathbf{f}(M_5, M_6, M_7) = f(1,0,0) = 1$

$P_8 = \mathbf{f}(M_9, M_{10}, M_{11}) = f(0,0,1) = 1$

P_1	P_2	P_4	P_8	
0	0	0	1	← Received check bits
1	0	1	1	← Re-computed check bits
1	0	1	0	Error Syndrom

Error Syndrom not **zero**, i.e, error in bit (1+4) = 5

Corrected codeword

P_1	P_2	M_3	P_4	M_5	M_6	M_7	P_8	M_9	M_{10}	M_{11}
0	0	1	0	0	0	0	1	0	0	1

Corrected message

1	0	0	0	0	0	1

Error Correction – Hamming code

- The **number** of check bits required **depends** on the number of message bits.
- Assume ***m*** message bits and ***r*** check bits in ***n***-bit **codewords**, i.e., ***n*** = (***m*** + ***r)***
- Each message has ***1*** **valid** codeword and ***n*** **invalid** codewords, that can be generated by **flipping** any **single bit** of the valid codeword.
- Each **message** has total (***n +1)*** **valid** and **invalid** codewords.
- There could be at most $\mathbf{2^m}$ messages by ***m***-bits.
- There could be at most (***n +1)**** $\mathbf{2^m}$ **valid** and **invalid** codewords
- There could be at most $\mathbf{2^n}$ **valid** and **invalid** codewords by ***n***-bits.
- Therefore $(n+1)* 2^m <= 2^n$ replacing ***n*** = (***m*** + ***r***)

$$(m + r + 1)* 2^m <= 2^{(m + r)}$$
$$<= 2^m * 2^r$$
$$(m + r + 1) <= 2^r$$

Error Correction – Hamming code

$(m + r + 1) <= 2^r$

7-bit message $m = 7$

$(7 + r + 1) <= 2^r$

r = 1, 2, 3 do not satisfy the above inequality
$r = 4$, i.e., **4** check bits and (***11***, ***7***) **Hamming Code**

Error Detection – Parity

Parity bit is added as the modulo 2 sum of data bits

- Equivalent to XOR; this is even parity
- Ex: 1110000 → 1110000 1, sum is 1
- Detection checks if the sum is wrong (an error)

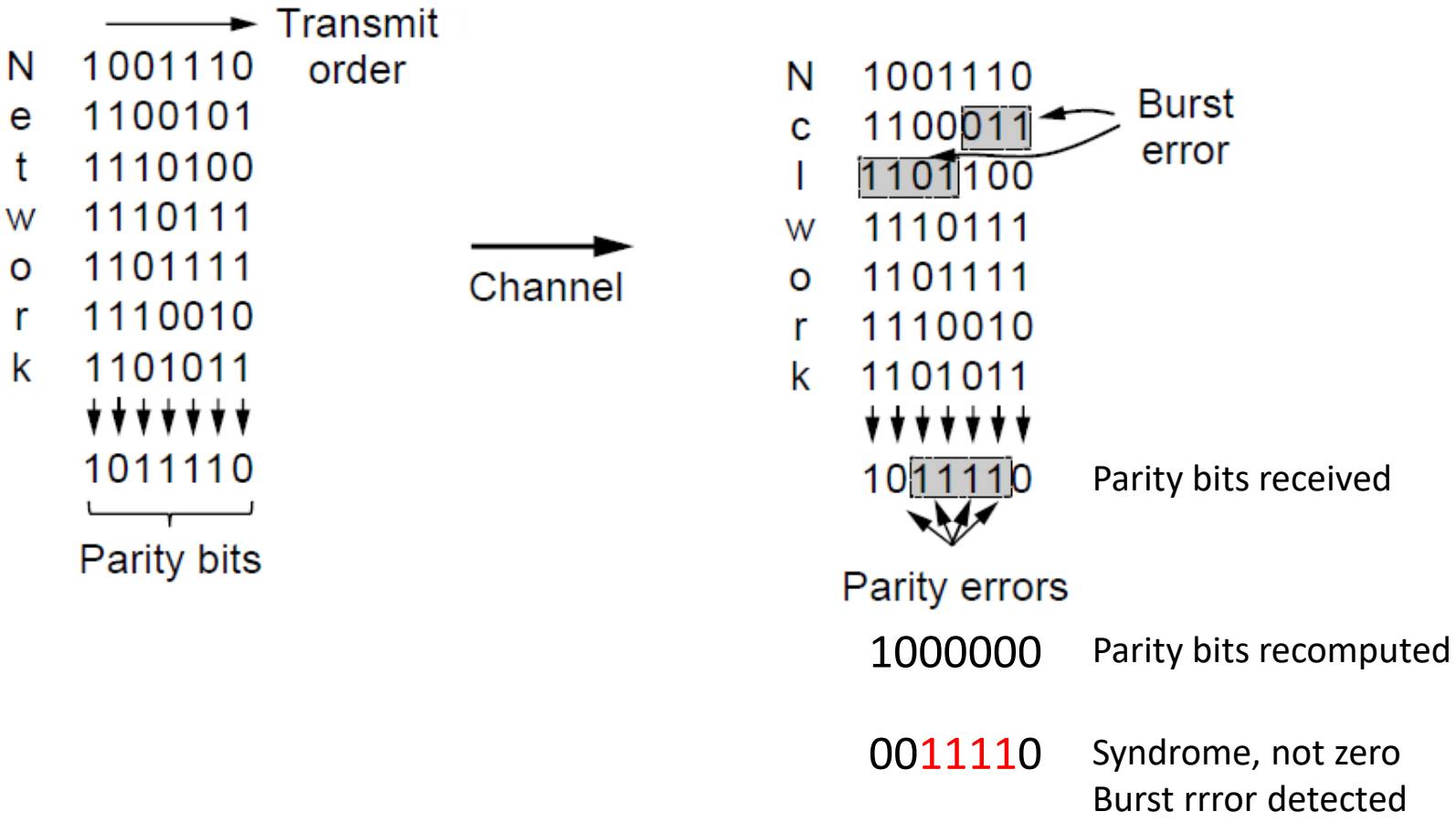
Simple way to detect an *odd* number of errors

- Ex: 1 error, 1110010 1; detected, sum 0 is wrong
- Ex: 3 errors, 1101100 1; detected sum 0 is wrong
- Ex: 2 errors, 1110110 1; *not detected*, sum 1 is right!
- Error can also be in the parity bit itself
- Random errors are detected with probability ½

Error Detection – Parity

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



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

Properties:

- Improved error detection over parity bits
- Detects bursts up to N errors
- Detects random errors with probability $(1 - \frac{1}{2^n})$

Error Detection – Checksums

IP Header Checksum Computation by Sender

20 Bytes IP Header content without checksum

4500 058c cadd 4000 ef06 0000 825f 808c 80d0 0297

4500
+058c
+cadd
+4000
+ef06
+0000
+825f
+808c
+80d0
+0297

3cac1
+3
---
cac4

One's complement of **cac4** is **353b**

20 Bytes IP Header content sent with checksum

4500 058c cadd 4000 ef06 353b 825f 808c 80d0 0297

Error Detection – Checksums

IP Header Checksum Computation by Receiver

20 Bytes IP Header content received with checksum

4500 058c cadd 4000 ef06 353b 825f 808c 80d0 0297

4500
+058c
+cadd
+4000
+ef06
+353b
+825f
+808c
+80d0
+0297

3fffc
+3
---
ffff

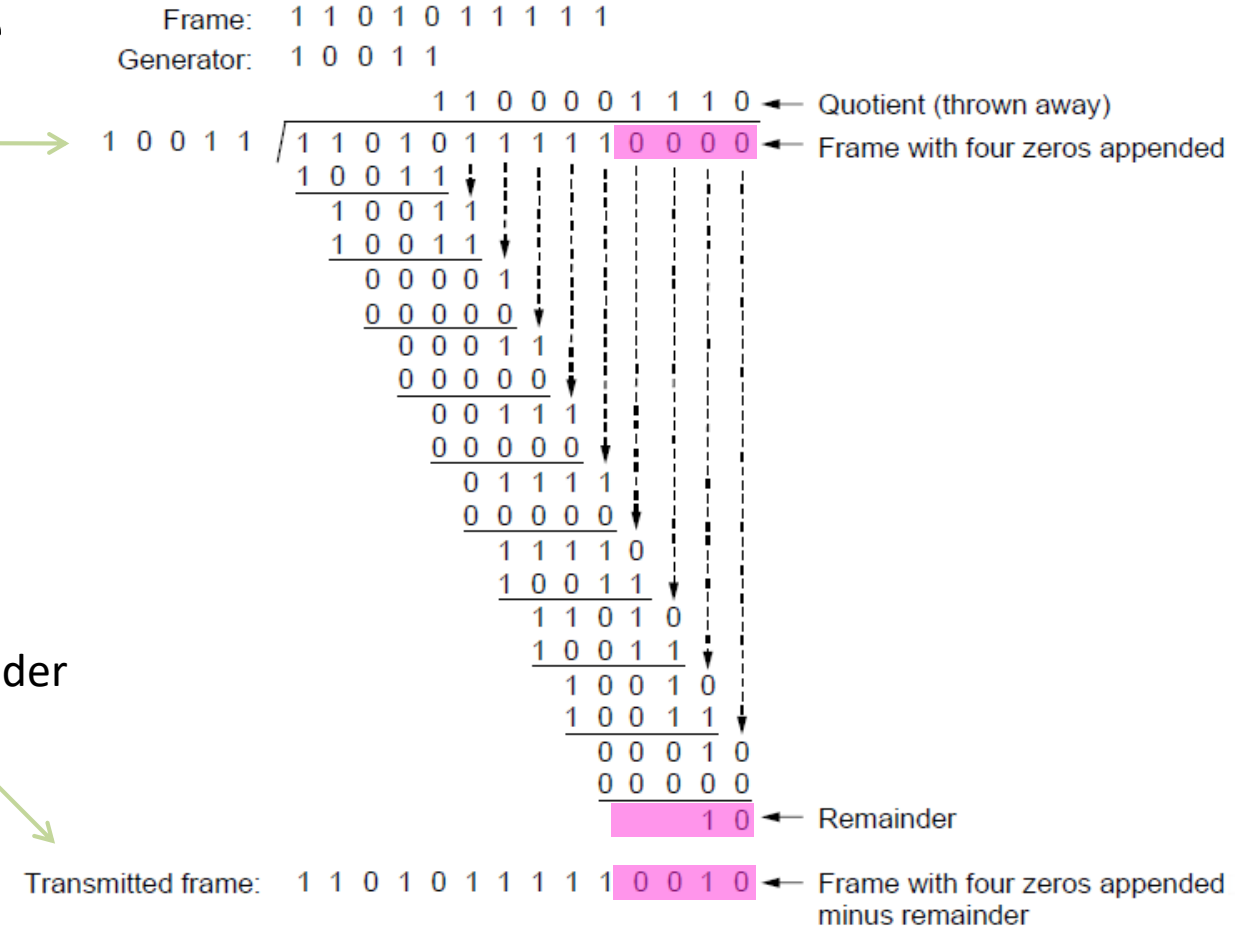
One's complement of **ffff** is **0000**

No errors in the header, if recomputed checksum is **zero**.

Error Detection – CRCs

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

Start by adding 0s to frame and try dividing



Offset by any reminder to make it evenly divisible

Error Detection – CRCs

Based on standard polynomials:

- Ex: Ethernet 32-bit CRC is defined by:

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

- Computed with simple shift/XOR circuits

Stronger detection than checksums:

- E.g., can detect all double bit errors
- Not vulnerable to systematic errors

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

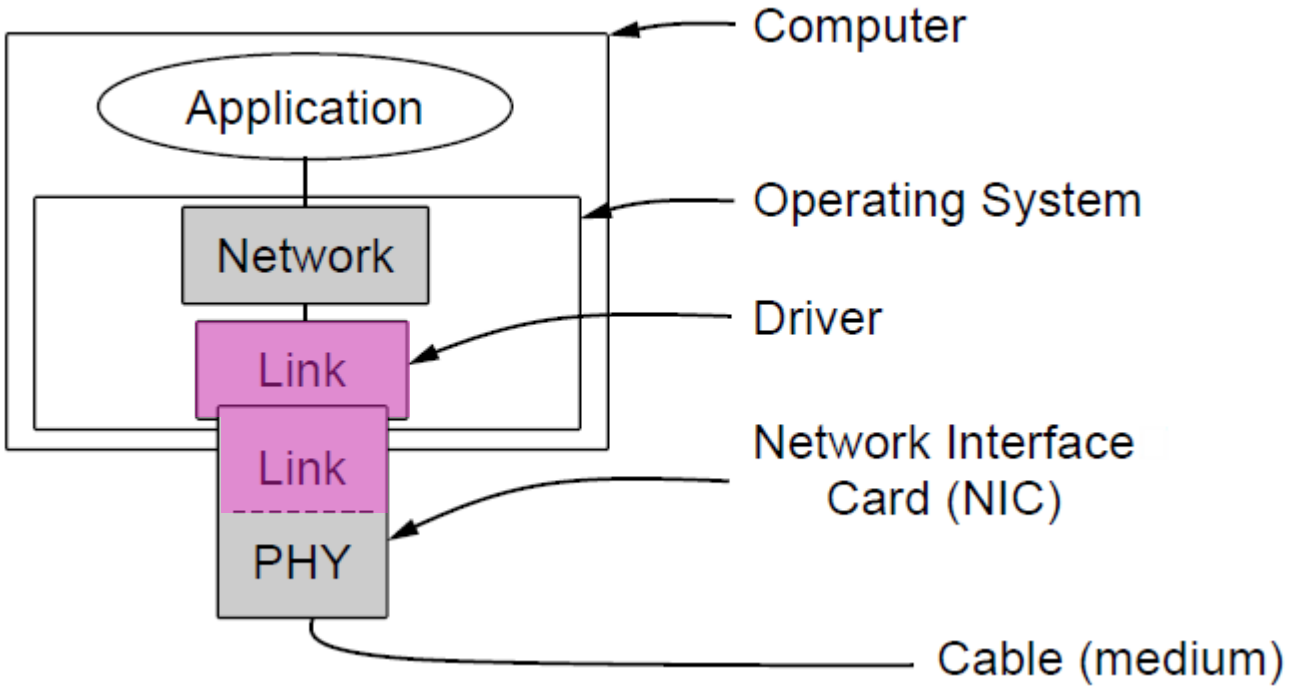
Flow control is a topic in both Link and Transport layers.

Elementary Data Link Protocols

- Link layer environment »
- Utopian Simplex Protocol »
- Stop-and-Wait Protocol for Error-free channel »
- Stop-and-Wait Protocol for Noisy channel »

Link layer environment

Commonly implemented as NICs and OS drivers: network layer (IP) is often OS



Link layer environment

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

```
#define MAX_PKT 1024

typedef unsigned int seq_nr;

typedef struct {
        unsigned char data[MAX_PKT];
} packet;

typedef enum {data, ack, nak} frame_kind;
```

Link layer environment

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

```
typedef struct {
        frame_kind kind;
        seq_nr seq;
        seq_nr ack;
        packet info;
} frame;
```

```
typedef enum {
        frame_arrival,
        CKsum_err,
        timeout,
        network_layer_ready,
        ack_timeout
} event_type;
```

Link layer environment

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

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

Utopian Simplex Protocol

An optimistic protocol (p1) to get us started

- Assumes no errors, and receiver as fast as sender
- Considers one-way data transfer

```
void sender1(void)
{
  frame s;
  packet buffer;

  while (true) {
       from_network_layer(&buffer);
       s.info = buffer;
       to_physical_layer(&s);
  }
}
```

Sender loops blasting frames

```
void receiver1(void)
{
  frame r;
  event_type event;

  while (true) {
       wait_for_event(&event);
       from_physical_layer(&r);
       to_network_layer(&r.info);
  }
}
```

Receiver loops eating frames

- That's it, no error or flow control …

Stop-and-Wait – Error-free channel

Protocol (p2) ensures sender can't outpace receiver:

- Receiver returns a dummy frame (ack) when ready
- Only one frame out at a time – called stop-and-wait
- We added flow control!

```
void sender2(void)
{
  frame s;
  packet buffer;
  event_type event;

  while (true) {
       from_network_layer(&buffer);
       s.info = buffer;
       to_physical_layer(&s);
       wait_for_event(&event);
  }
}
```

Sender waits to for ack after passing frame to physical layer

```
void receiver2(void)
{
  frame r, s;
  event_type event;
  while (true) {
       wait_for_event(&event);
       from_physical_layer(&r);
       to_network_layer(&r.info);
       to_physical_layer(&s);
  }
}
```

Receiver sends ack after passing frame to network layer

Stop-and-Wait – Noisy channel

ARQ (Automatic Repeat reQuest) adds error control

- Receiver acks frames that are correctly delivered
- Sender sets timer and resends frame if no ack)

For correctness, frames and acks must be numbered

- Else receiver can't tell retransmission (due to lost ack or early timer) from new frame
- For stop-and-wait, 2 numbers (1 bit) are sufficient

Stop-and-Wait – Noisy channel

Sender loop (p3):

```
void sender3(void) {
  seq_nr next_frame_to_send;
  frame s;
  packet buffer;
  event_type event;

  next_frame_to_send = 0;
  from_network_layer(&buffer);
  while (true) {
      s.info = buffer;
      s.seq = next_frame_to_send;
      to_physical_layer(&s);
      start_timer(s.seq);
      wait_for_event(&event);
      if (event == frame_arrival) {
          from_physical_layer(&s);
          if (s.ack == next_frame_to_send) {
              stop_timer(s.ack);
              from_network_layer(&buffer);
              inc(next_frame_to_send);
          }
      }
  }
}
```

Send frame (or retransmission)

Set timer for retransmission

Wait for ack or timeout

If a good ack then set up for the next frame to send (else the old frame will be retransmitted)

Stop-and-Wait – Noisy channel

Receiver loop (p3):

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

  frame_expected = 0;
  while (true) {
      wait_for_event(&event);
      if (event == frame_arrival) {
          from_physical_layer(&r);
          if (r.seq == frame_expected) {
              to_network_layer(&r.info);
              inc(frame_expected);
          }
          s.ack = 1 – frame_expected;
          to_physical_layer(&s);
      }
  }
}
```

Wait for a frame

If it's new then take it and advance expected frame

Ack current frame

Sliding Window Protocols

- Sliding Window concept »
- One-bit Sliding Window »
- Go-Back-N »
- Selective Repeat »

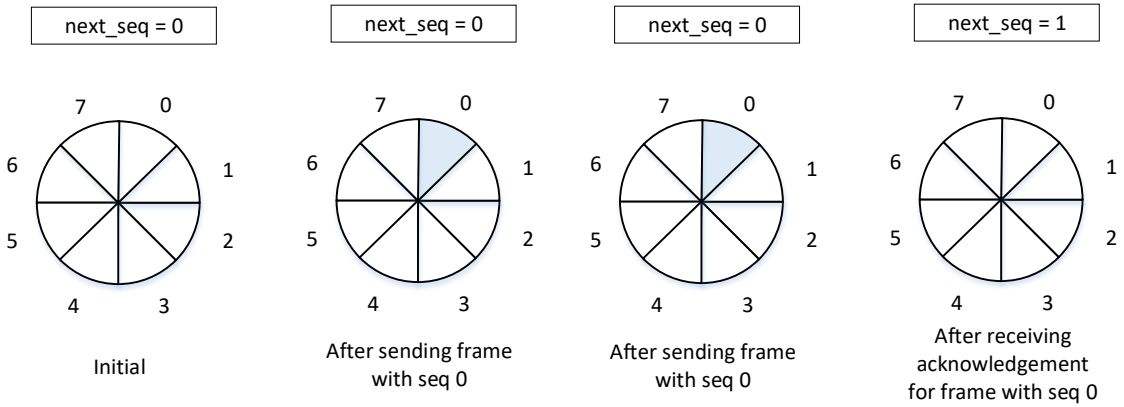
Sliding Window concept

- Sender maintains window of frames it can send
  - Needs to buffer them for possible retransmission
  - Window advances with next acknowledgements
- Receiver maintains window of frames it can receive
  - Needs to keep buffer space for arrivals
  - Window advances with in-order arrivals

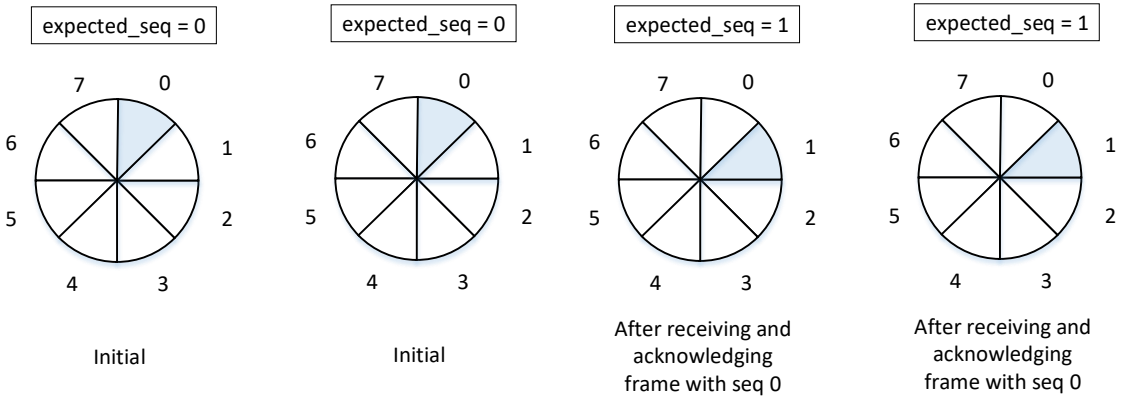
Sliding Window concept

Sequence number 3-bit, Window size is 1

Sender



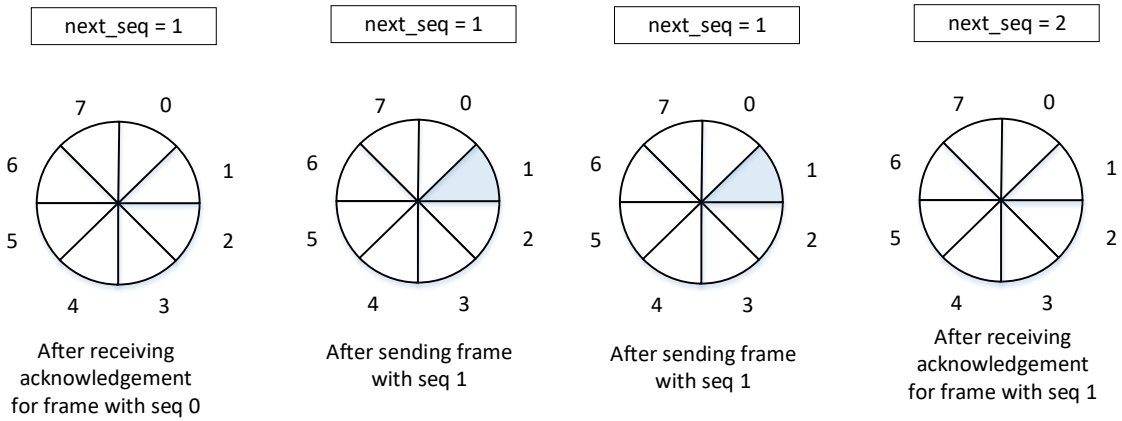
Receiver



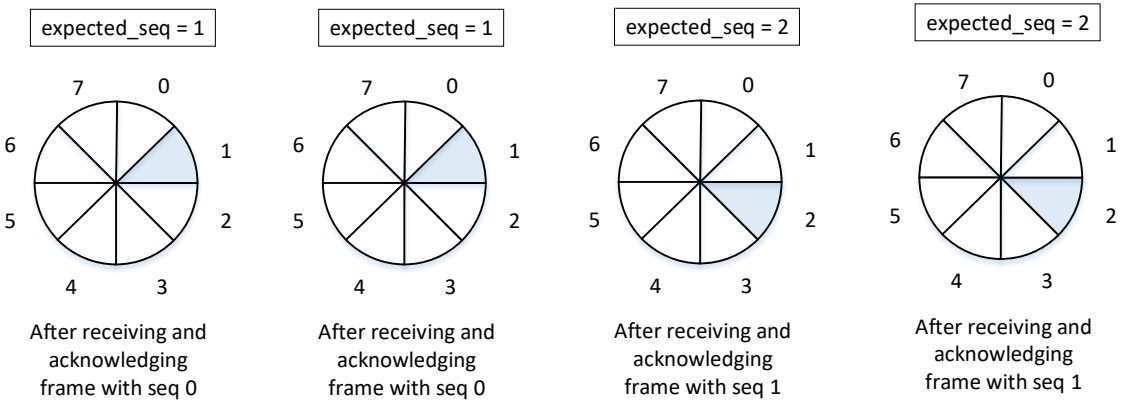
Sliding Window concept

Sequence number 3-bit, Window size is 1

Sender



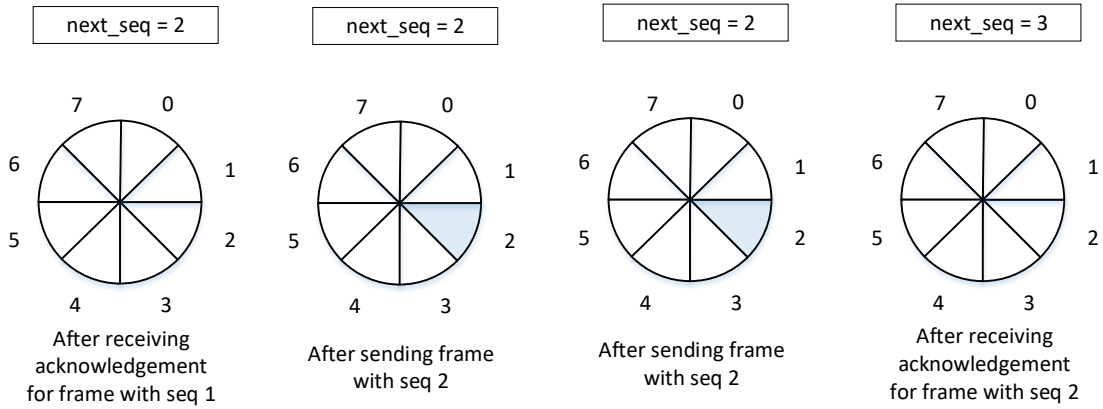
Receiver



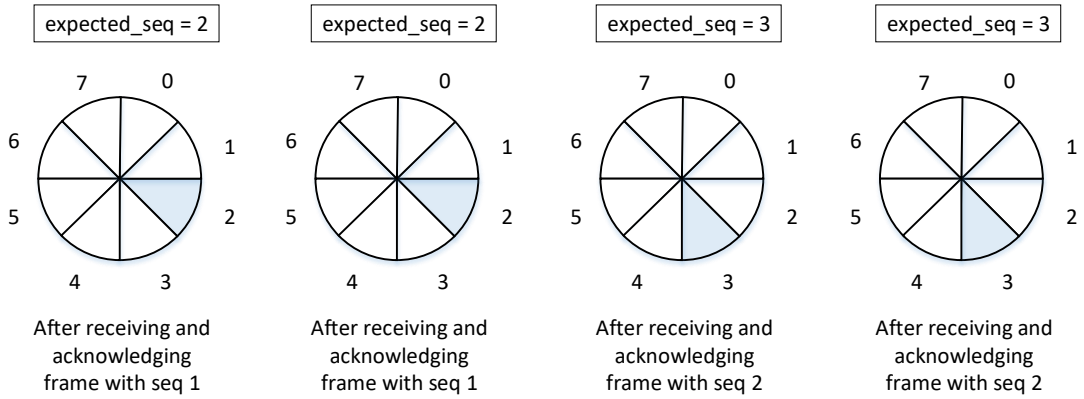
Sliding Window concept

Sequence number 3-bit, Window size is 1

Sender



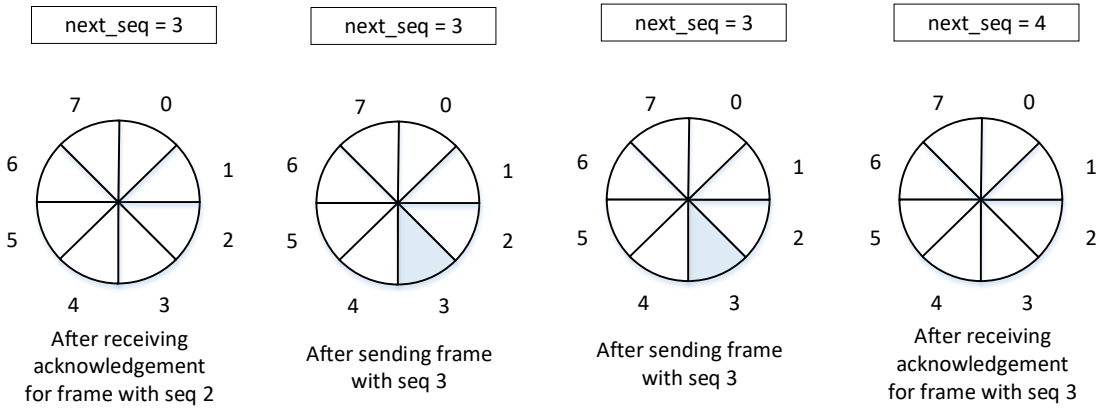
Receiver



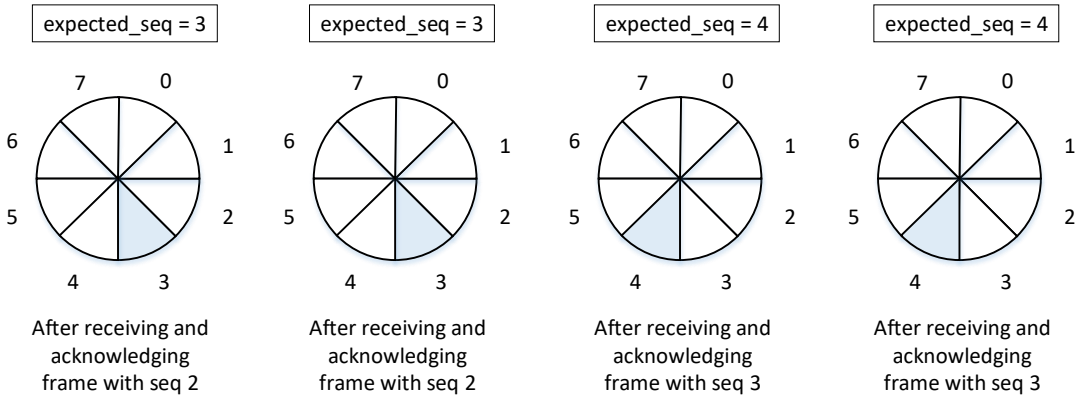
Sliding Window concept

Sequence number 3-bit, Window size is 1

Sender



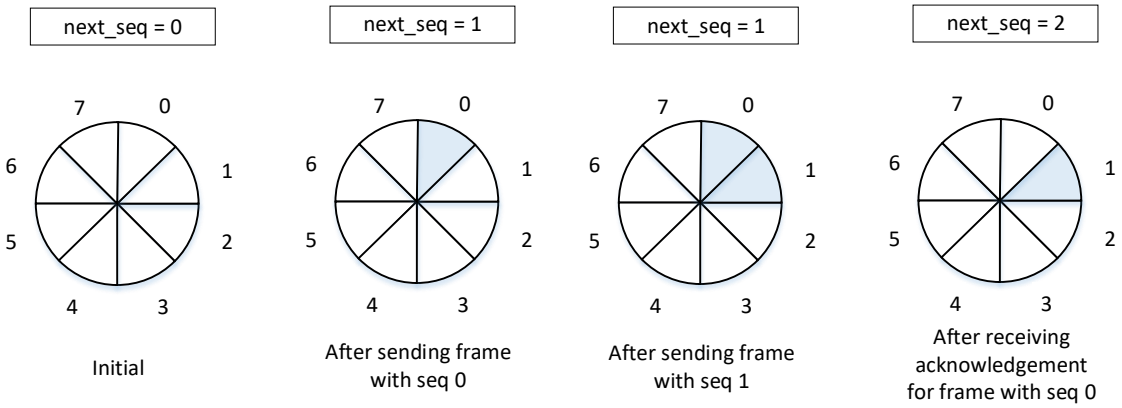
Receiver



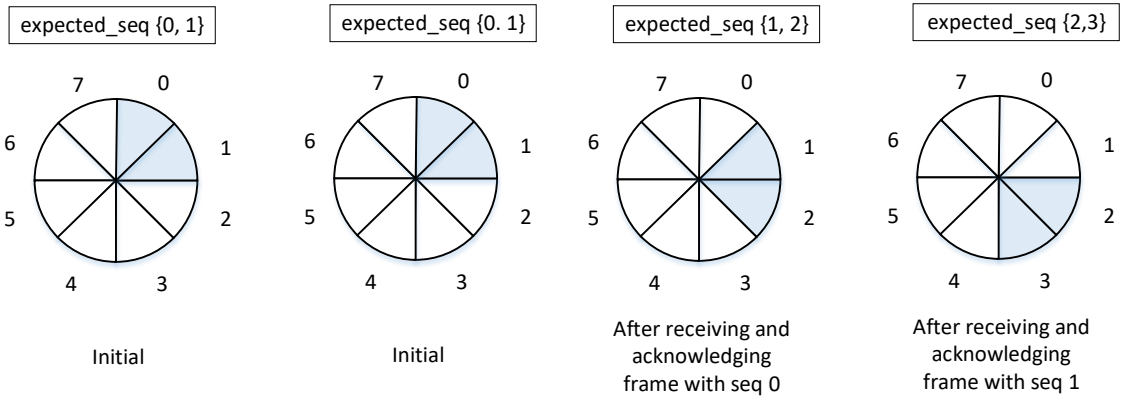
Sliding Window concept

Sequence number 3-bit, Window size is 2

Sender



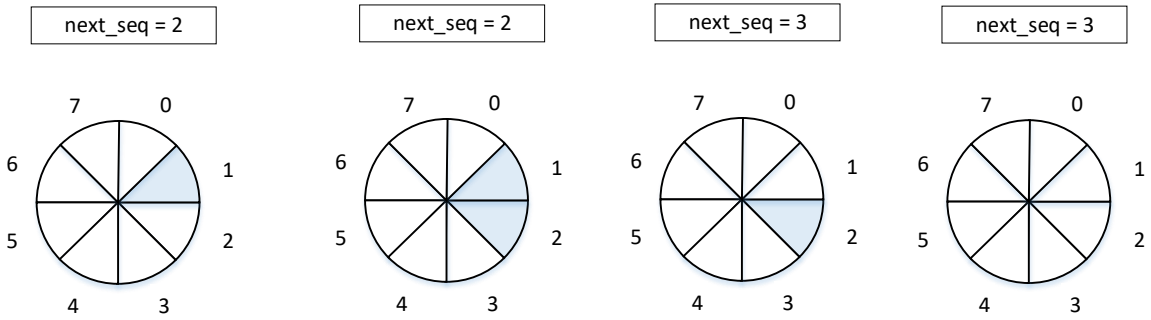
Receiver



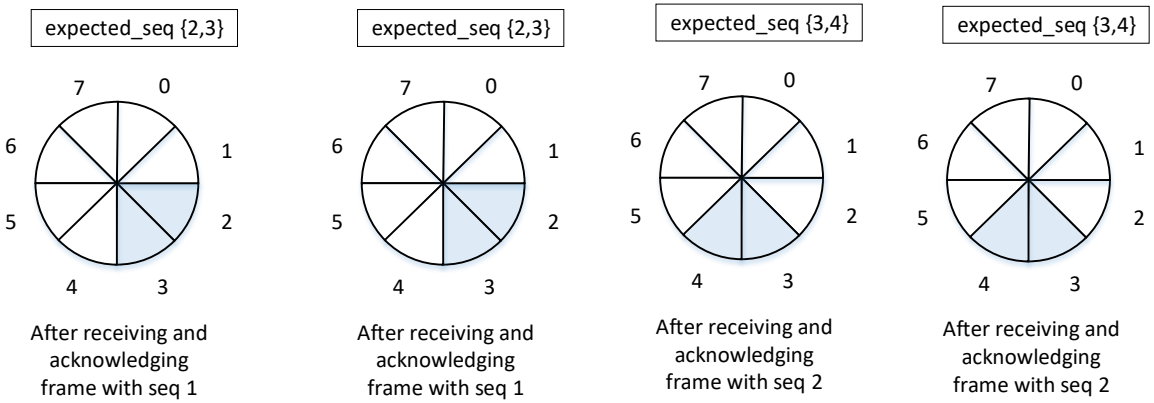
Sliding Window concept

Sequence number 3-bit, Window size is 2

Sender



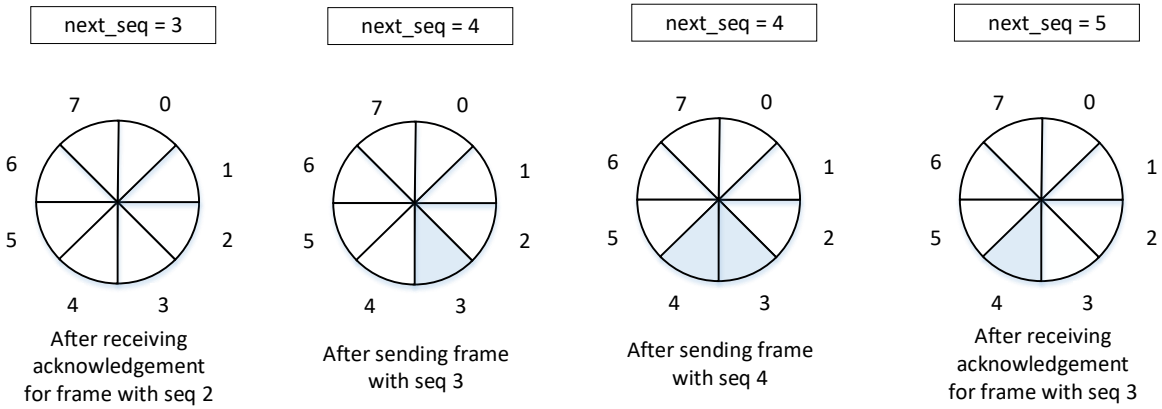
Receiver



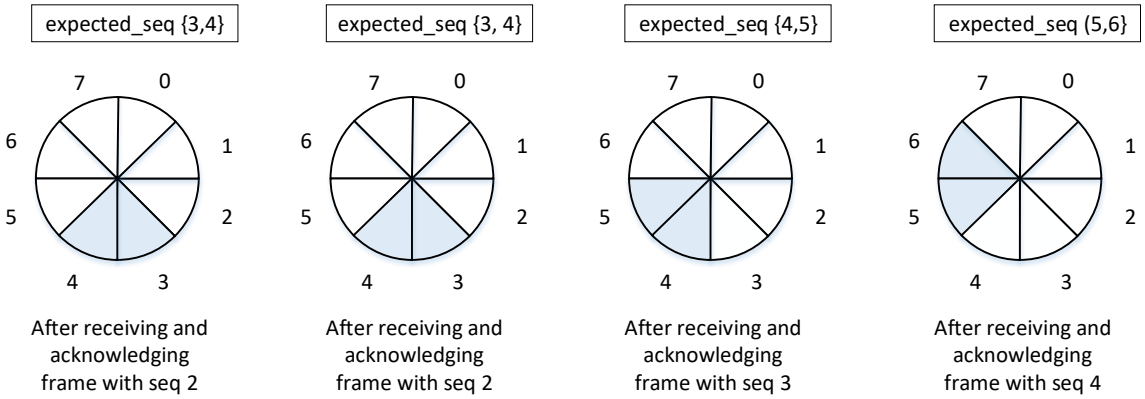
Sliding Window concept

Sequence number 3-bit, Window size is 2

Sender



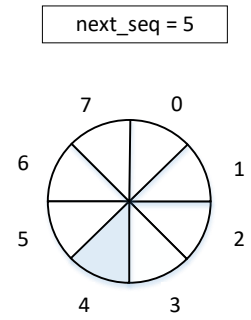
Receiver



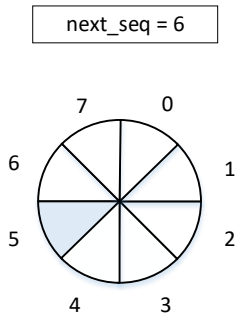
Sliding Window concept

Sequence number 3-bit, Window size is 2

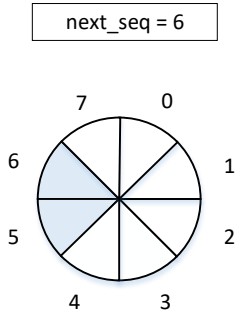
Sender



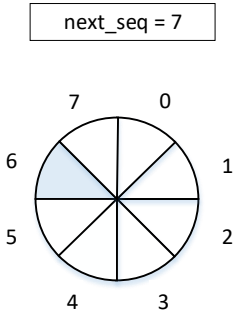
After receiving acknowledgement for frame with seq 3



After sending frame with seq 5 and receiving ackknowledgement for 4

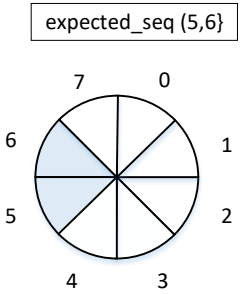


After sending frame with seq 6

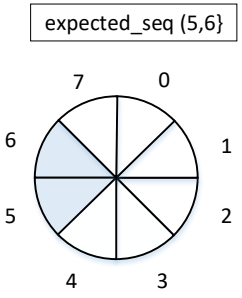


After receiving acknowledgement for frame with seq 5

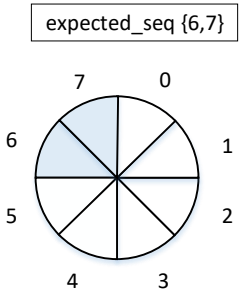
Receiver



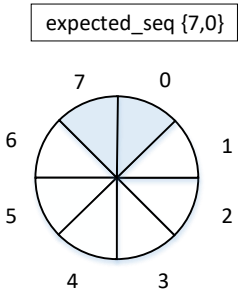
After receiving and acknowledging frame with seq 4



After receiving and acknowledging frame with seq 4



After receiving and acknowledging frame with seq 5

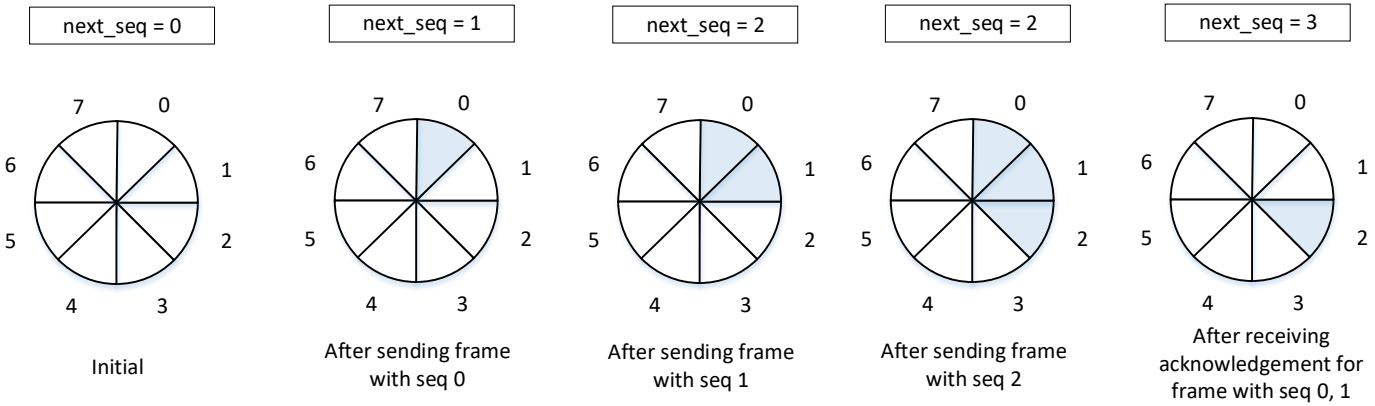


After receiving and acknowledging frame with seq 6

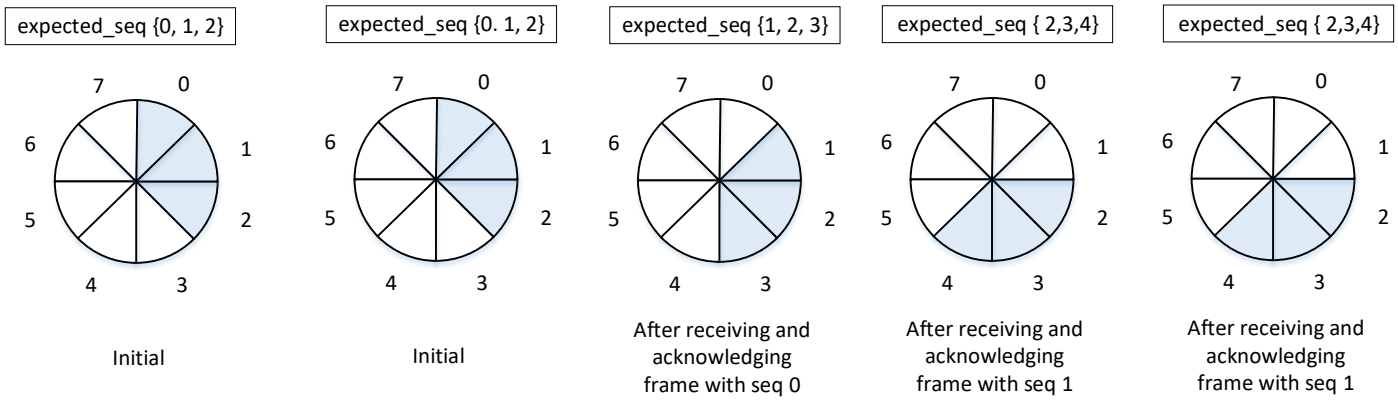
Sliding Window concept

Sequence number 3-bit, Window size is 3

Sender



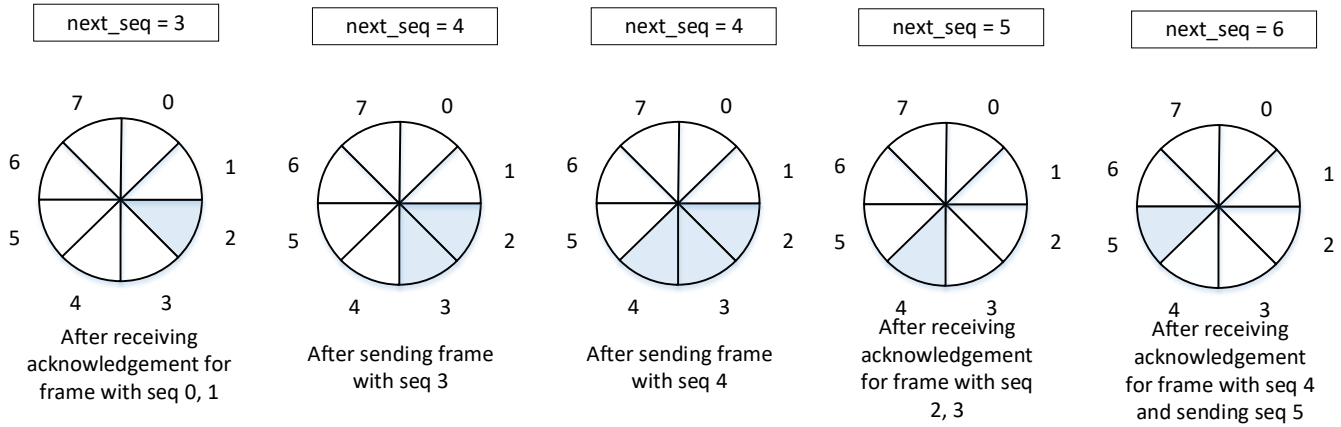
Receiver



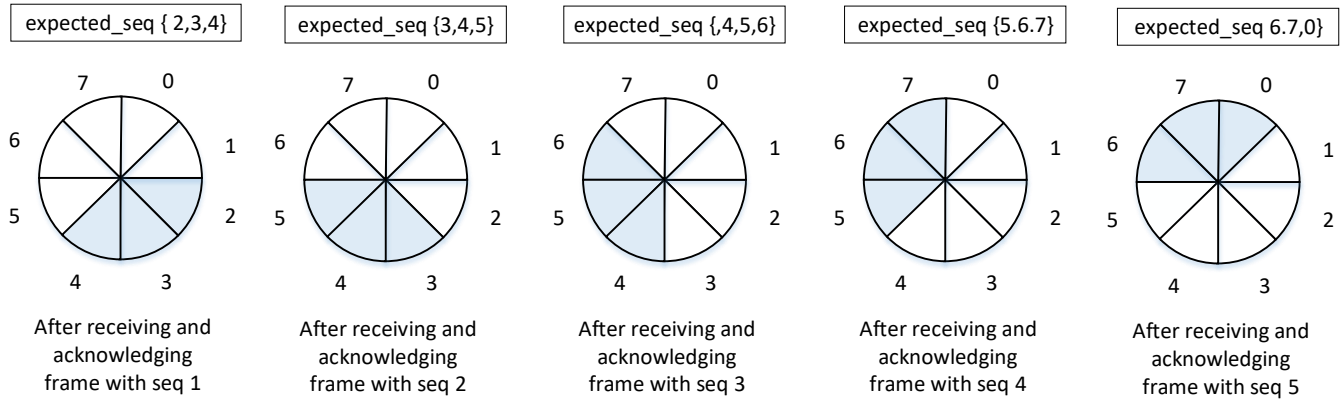
Sliding Window concept

Sequence number 3-bit, Window size is 3

Sender



Receiver



Sliding Window concept

Larger windows enable pipelining for efficient link use

- Stop-and-wait (w=1) is inefficient for long links
- Best **window** (***w***) depends on one-way **bandwidth-delay** (***BD***) of the link and **frame size** (***F***).
- We want to use $w_{max} = 2BD/F+1$ to ensure high link utilization
- Otherwise, link utilization of using any other window size (w_a) is computed as (w_a/w_{max})
- For example, **1Mbps** link with **100ms** delay and **2kb** frame can use **window size 101** to ensure high utilization.
- But using a **window size *16*** on the same link give us 16/101 = ***0.158*** link **utilization**.

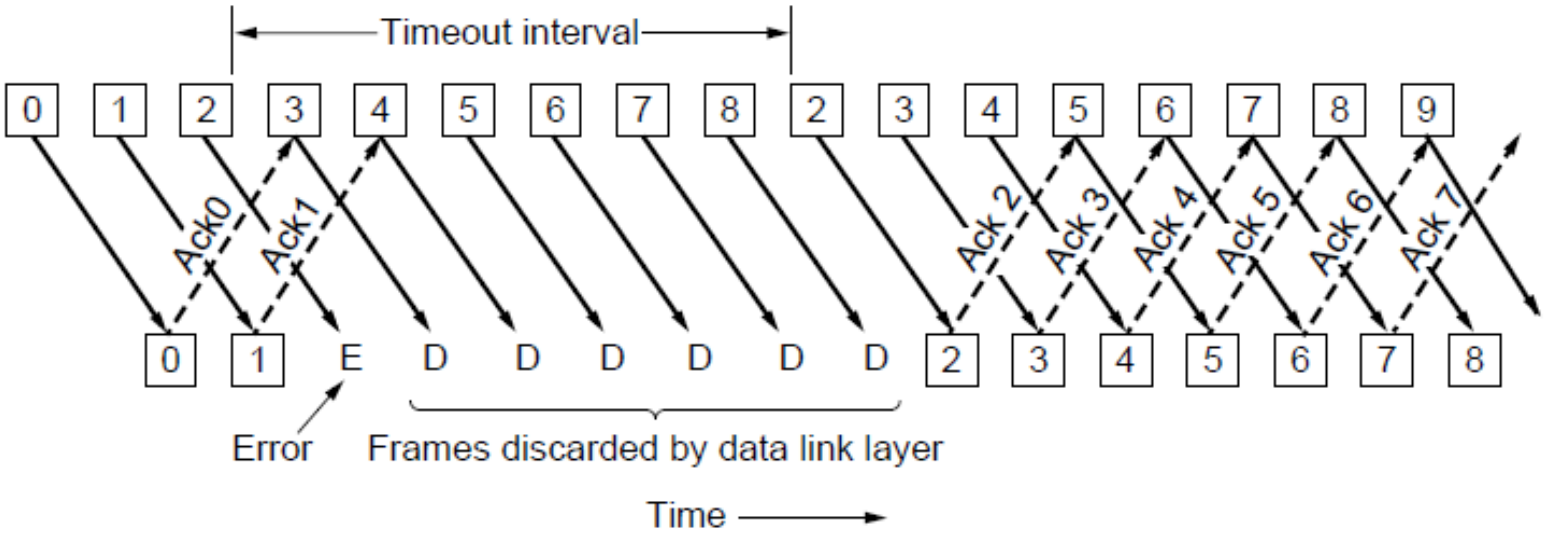
Sliding Window concept

Pipelining leads to different choices for errors/buffering

- We will consider **Go-Back-N** and **Selective Repeat**
- Go-Back-N Receiver only accepts/acks frames that arrive in order, i.e., receive window size 1 is sufficient
- Selective Repeat Receiver accepts frames anywhere in receive window, i.e., receive window size must be equal to send window size.

Go-Back-N

- Discards frames that follow a missing/errored frame
- Sender times out and resends all outstanding frames



Go-Back-N

Max window size with respect to **sequence number**

- With **n**-bit **sequence numbers**, sequence numbers **range** is **0** to (**2^n-1**) and max sequence number is (**2^n-1**)
- For example, for 3-bit sequence numbers, sequence numbers range is **0** to (**2^3-1**) or **0** to **7** (0,1,2,3,4,5,6,7) and the max sequence number is **7**

Go-Back-N

Max window size with respect to **sequence number**

- In Go-Back-N if the sender is allowed to use the maximum possible window size, i.e., $\mathbf{2^n}$ or **8** (in above scenario), it will be allowed to send 8 frames using all the sequence numbers, i.e., 0,1,2,3,4,5,6, and 7
- Assume receiver has received and acknowledged 8 frames but all the acknowledgements get lost.
- Sender times out for not receiving the acknowledgments and retransmits 8 frames successively with the sequence numbers 0,1,2,3,4,5,6, and 7.

Go-Back-N

Max window size with respect to **sequence number**

- Receiver incorrectly assumes 8 new frame arrivals instead of frame retransmissions as it is expecting a new frame with sequence number 0, then 1, and so on, i.e., the protocol fails.
- If the sender is allowed to use one less than the maximum possible window size, i.e., $\mathbf{2^n - 1}$ or **7** (in above scenario), it will be allowed to send 7 frames using the sequence numbers, i.e., 0,1,2,3,4,5, and 6
- If receiver has received and acknowledged 7 frames but all the acknowledgements get lost.

Go-Back-N

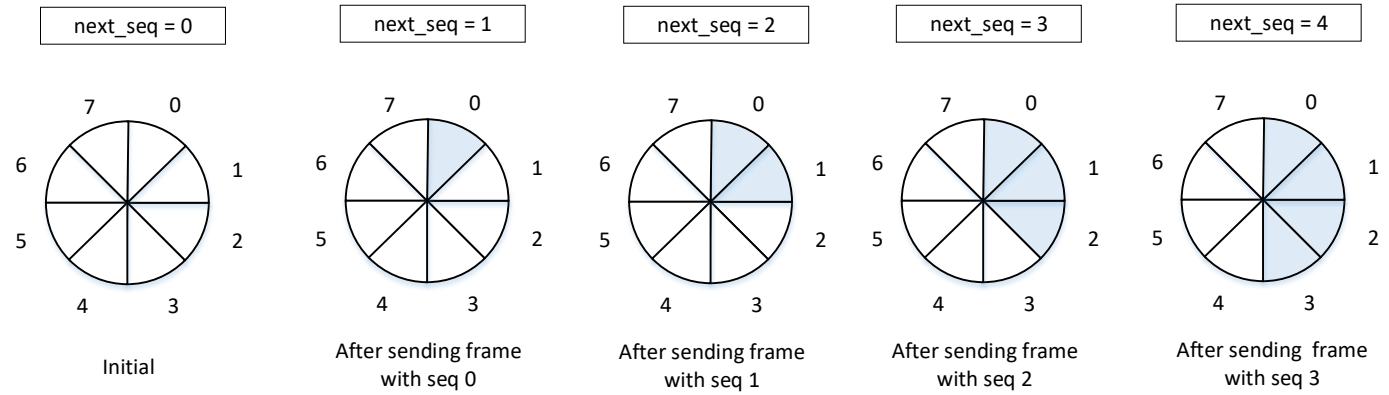
Max window size with respect to **sequence number**

- Sender times out for not receiving the acknowledgments and retransmits 7 frames successively with the sequence numbers 0,1,2,3,4,5, and 6.
- Receiver will not assumes 7 new frame arrivals instead of frame retransmissions as it is expecting a new frame with sequence number 7 not 0 and so on, i.e., the protocol succeeds.
- For this reason, sender window size in Go-Back-N is ($\mathbf{2^n - 1}$) with **n**-bit **sequence numbers**.

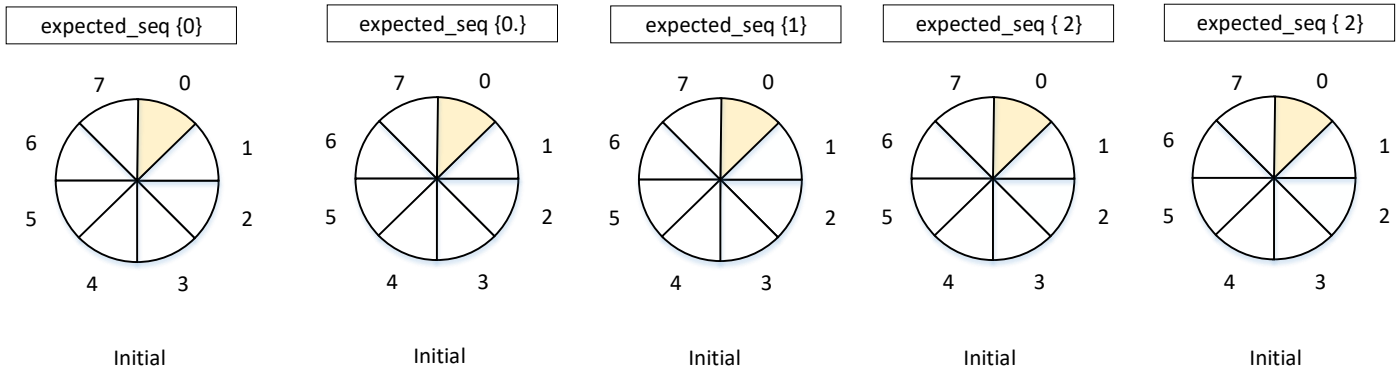
Go-Back-N

Sequence number 3-bit, Window size, Sender 7 Receiver 1

Sender



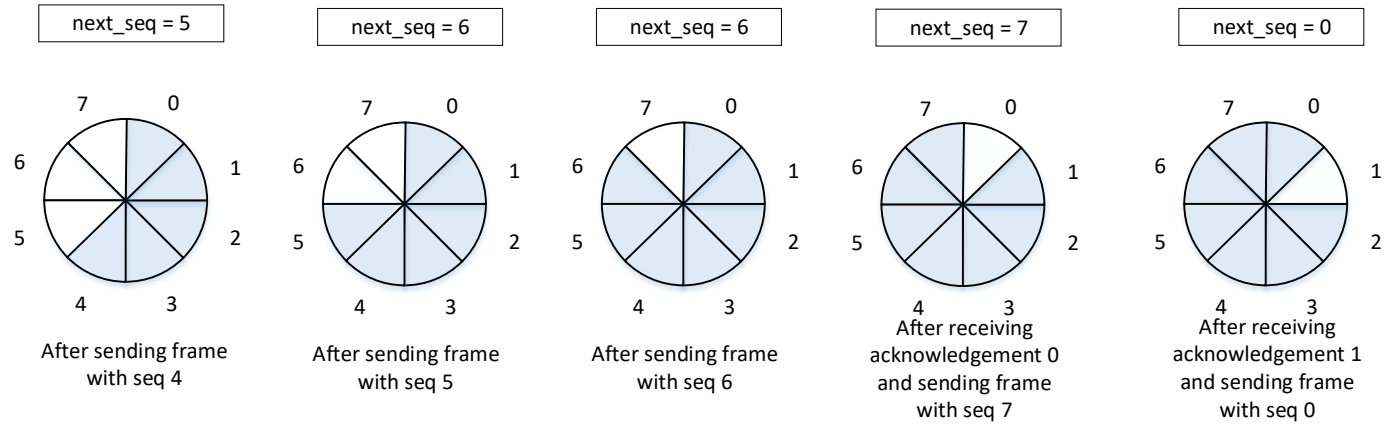
Receiver



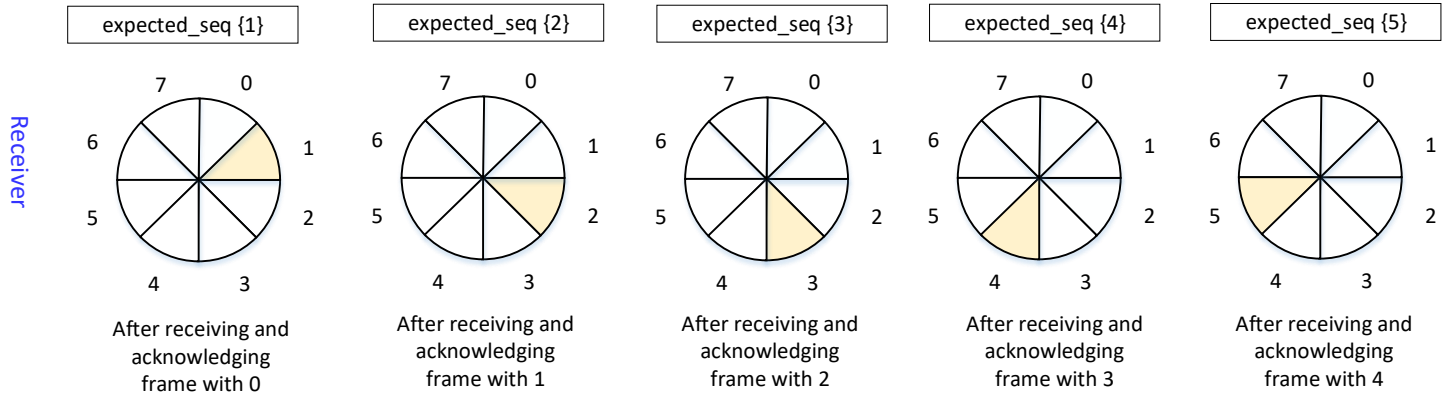
Go-Back-N

Sequence number 3-bit, Window size, Sender 7 Receiver 1

Sender



Receiver



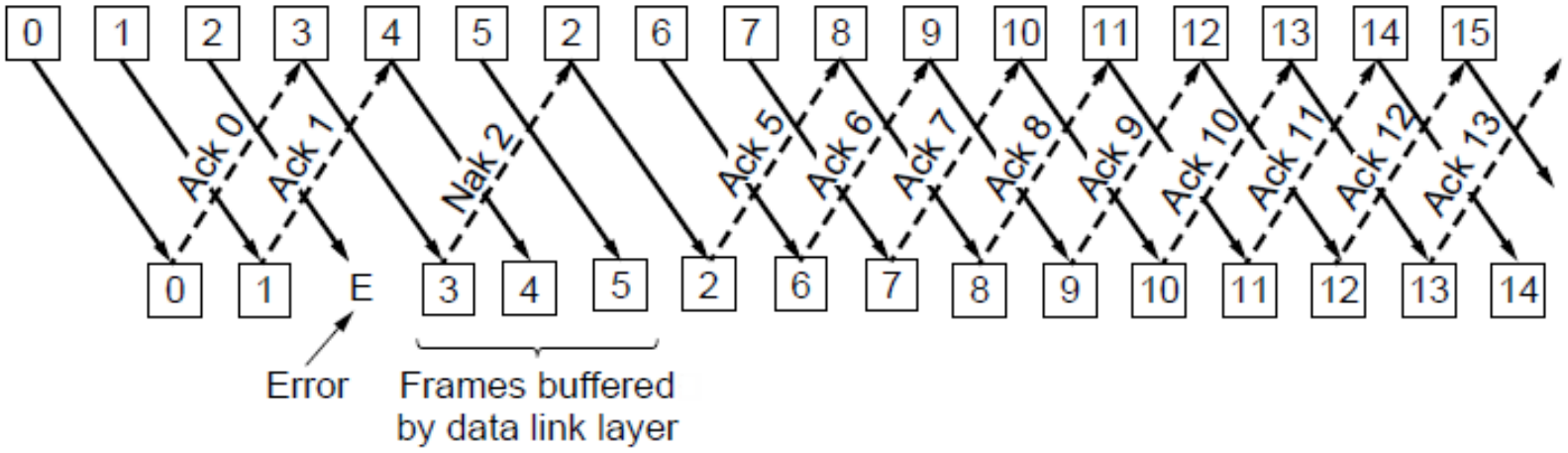
Go-Back-N

- Tradeoff made for Go-Back-N:
  - Simple strategy for receiver; needs only 1 frame
  - Wastes link bandwidth for errors with large windows; entire window is retransmitted

- Implemented as p5 (see code in book)

Selective Repeat

- Cumulative ack indicates highest in-order frame
- NAK (negative ack) causes sender retransmission of a missing frame before a timeout resends window



Selective Repeat

Max window size with respect to **sequence number**

- In Selective Repeat if the sender is allowed to use one less than the maximum possible window size, i.e., ($\mathbf{2^n}$ -1) or **7** (in above scenario) as in Go-Back-N, it will be allowed to send 7 frames using all the sequence numbers, i.e., 0,1,2,3,4,5,and 6
- Assume receiver has received and acknowledged 7 frames but all the acknowledgements get lost.

Selective Repeat

Max window size with respect to **sequence number**

- Sender times out for not receiving the acknowledgments and retransmits 7 frames successively with the sequence numbers 0,1,2,3,4,5, and 6.
- Receiver incorrectly assumes 6 new frame arrivals instead of frame retransmissions as it is expecting new frames with sequence number 7,0,1,2,3,4,5, i.e., the protocol fails.
- If the sender is allowed to use only the half of the maximum possible window size, i.e., $\mathbf{2^{n-1}}$ or **4** (in above scenario), it will be allowed to send only 4 frames using the sequence numbers, i.e., 0,1,2,and 3
- If receiver has received and acknowledged 4 frames but all the acknowledgements get lost.

Selective Repeat

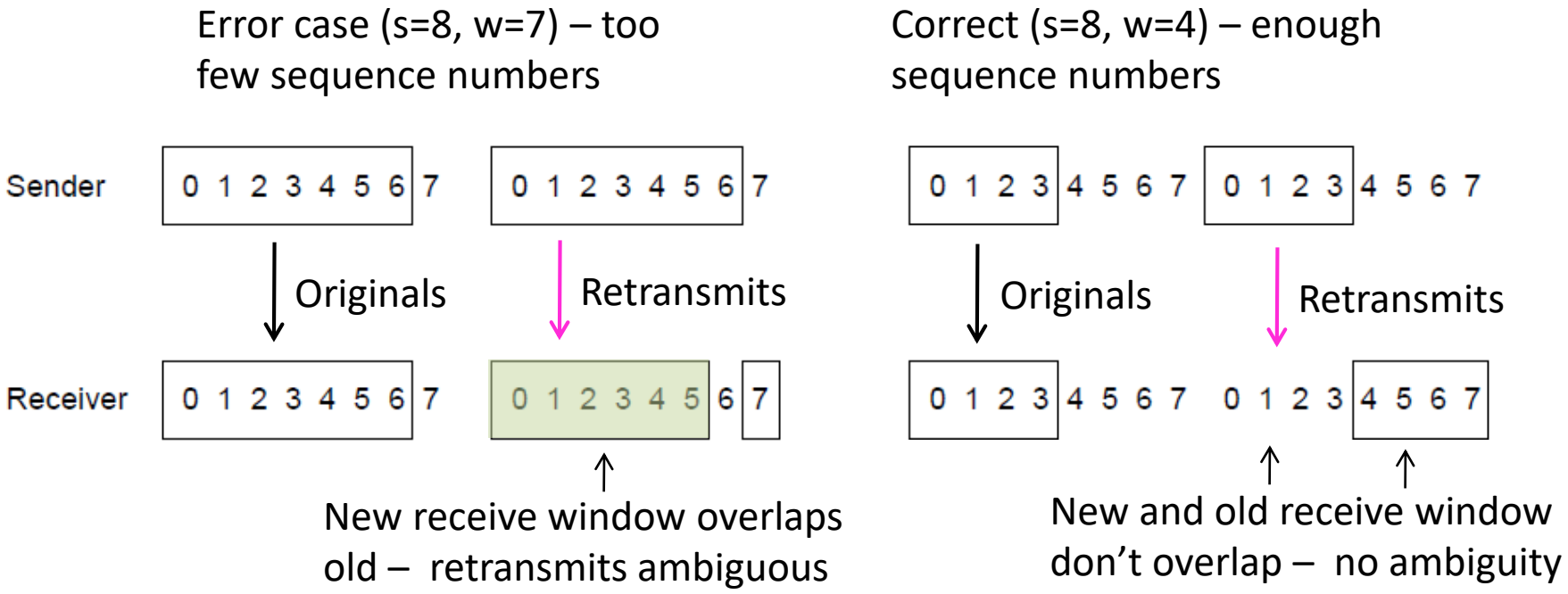
Max window size with respect to **sequence number**

- Sender times out for not receiving the acknowledgments and retransmits 4 frames successively with the sequence numbers 0,1,2, and 3.
- Receiver will not assumes 4 new frame arrivals instead of frame retransmissions as it is expecting the new frames with sequence numbers 4,5,6,and 7 not 0,1,2, and 3, i.e., the protocol succeeds.
- For this reason, sender window size in Selective Repeat is ($\mathbf{2^{n-1}}$) with **n**-bit **sequence numbers**.

Selective Repeat

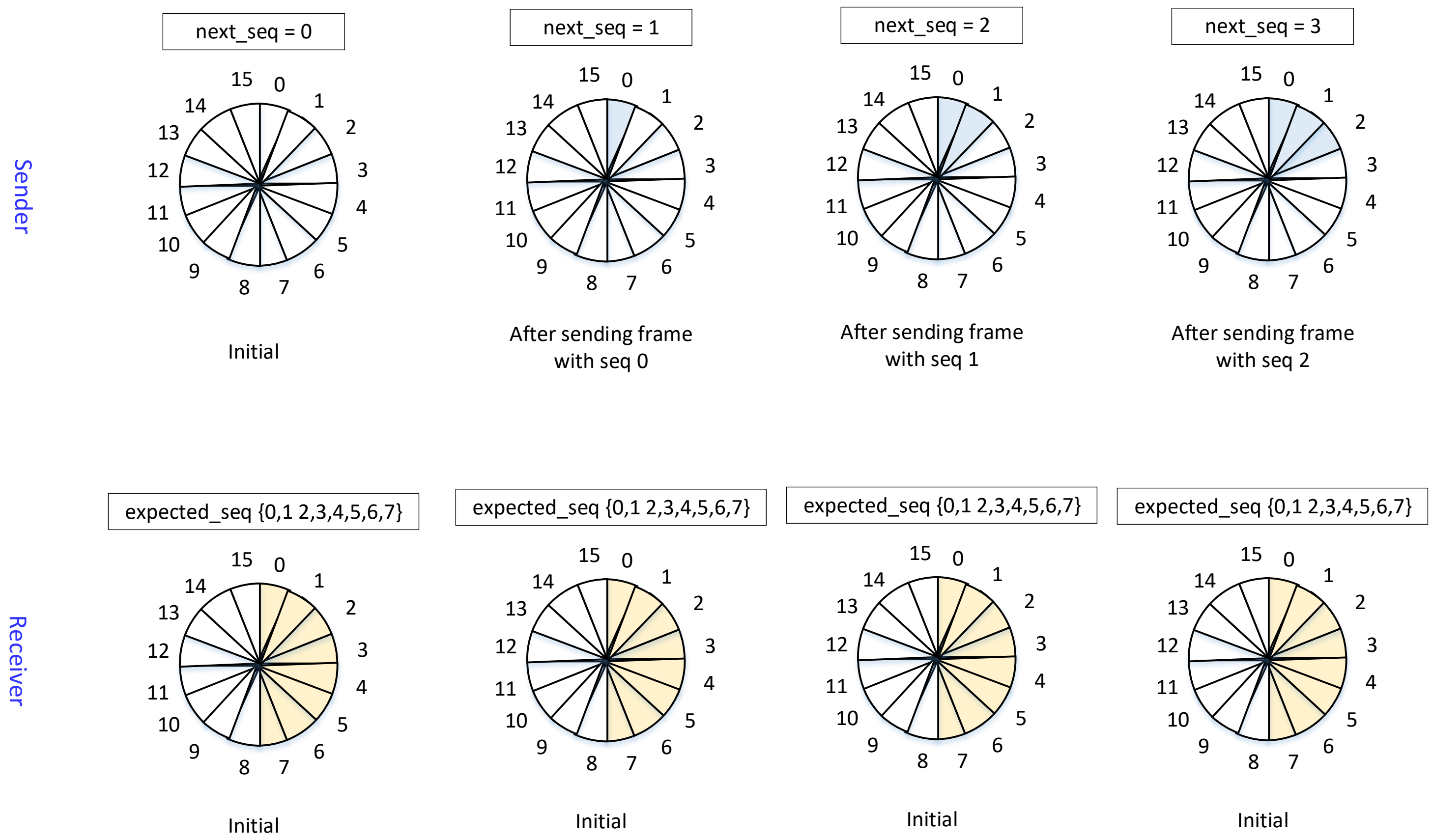
For correctness, we require:

- Sequence numbers (s) at least twice the window (w)



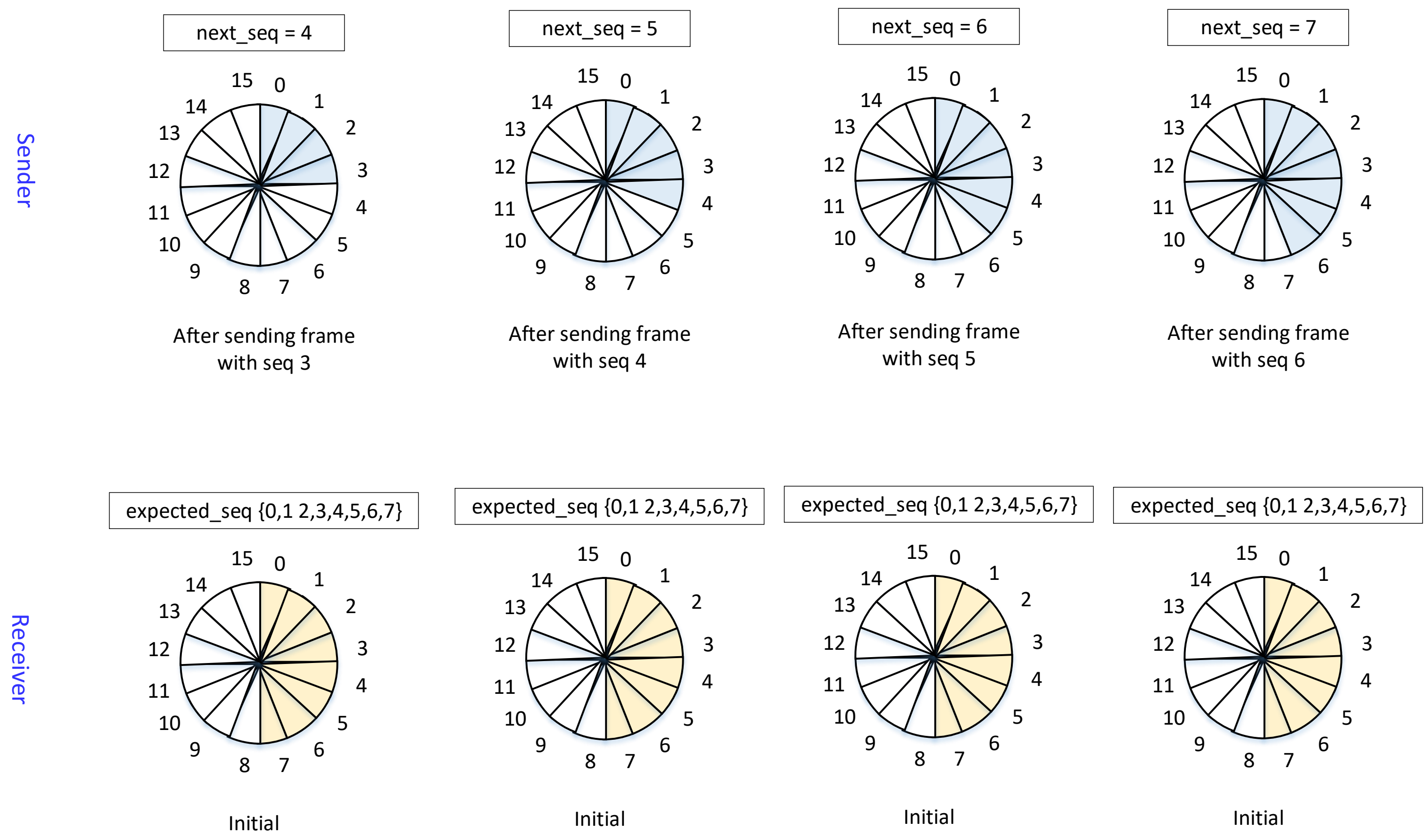
Selective Repeat

Sequence number 4-bit, Window size, Sender 8 Receiver 8



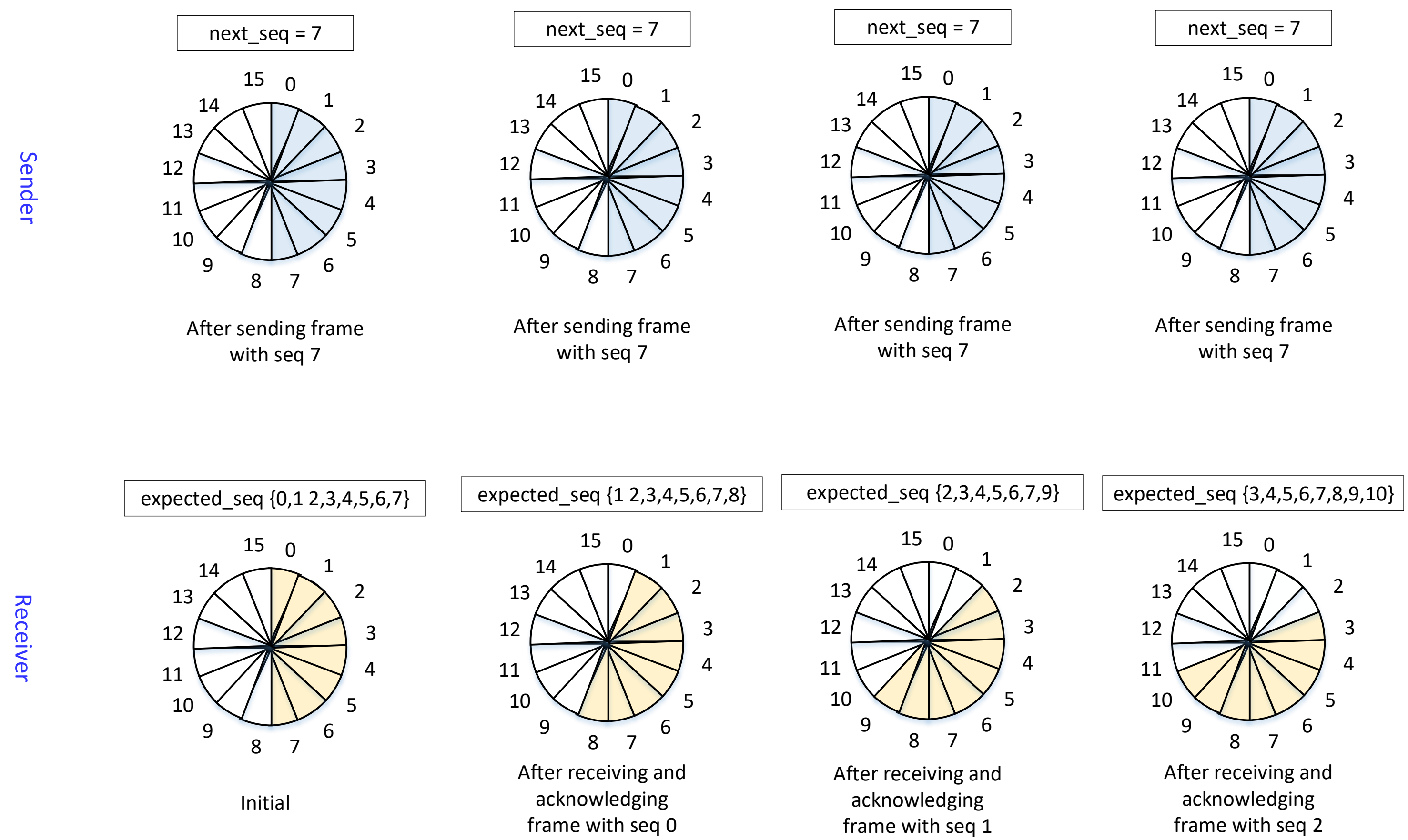
Selective Repeat

Sequence number 4-bit, Window size, Sender 8 Receiver 8



Selective Repeat

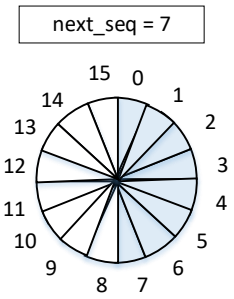
Sequence number 4-bit, Window size, Sender 8 Receiver 8



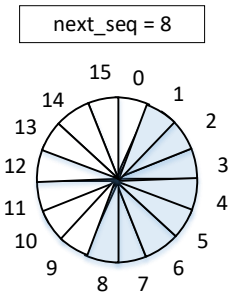
Selective Repeat

Sequence number 4-bit, Window size, Sender 8 Receiver 8

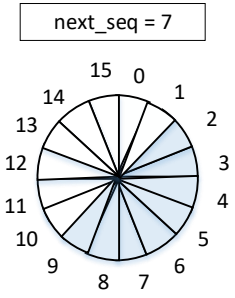
Sender



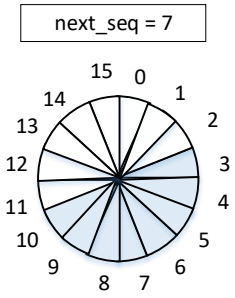
After sending frame with seq 7



After receiving acknowledgement 0 and sending frame with seq 8

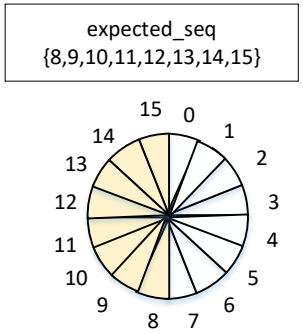


After receiving acknowledgement 1 and sending frame with seq 9

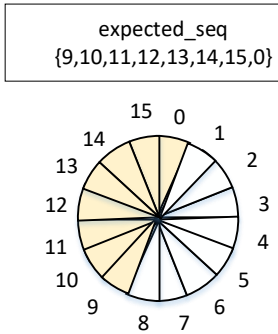


After receiving acknowledgement 2 and sending frame with seq 10

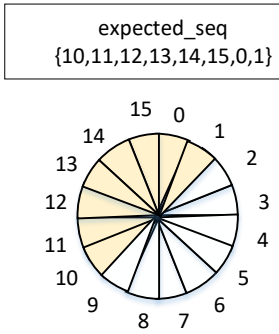
Receiver



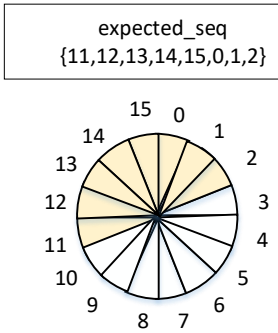
After receiving and acknowledging frame with seq 7



After receiving and acknowledging frame with seq 8



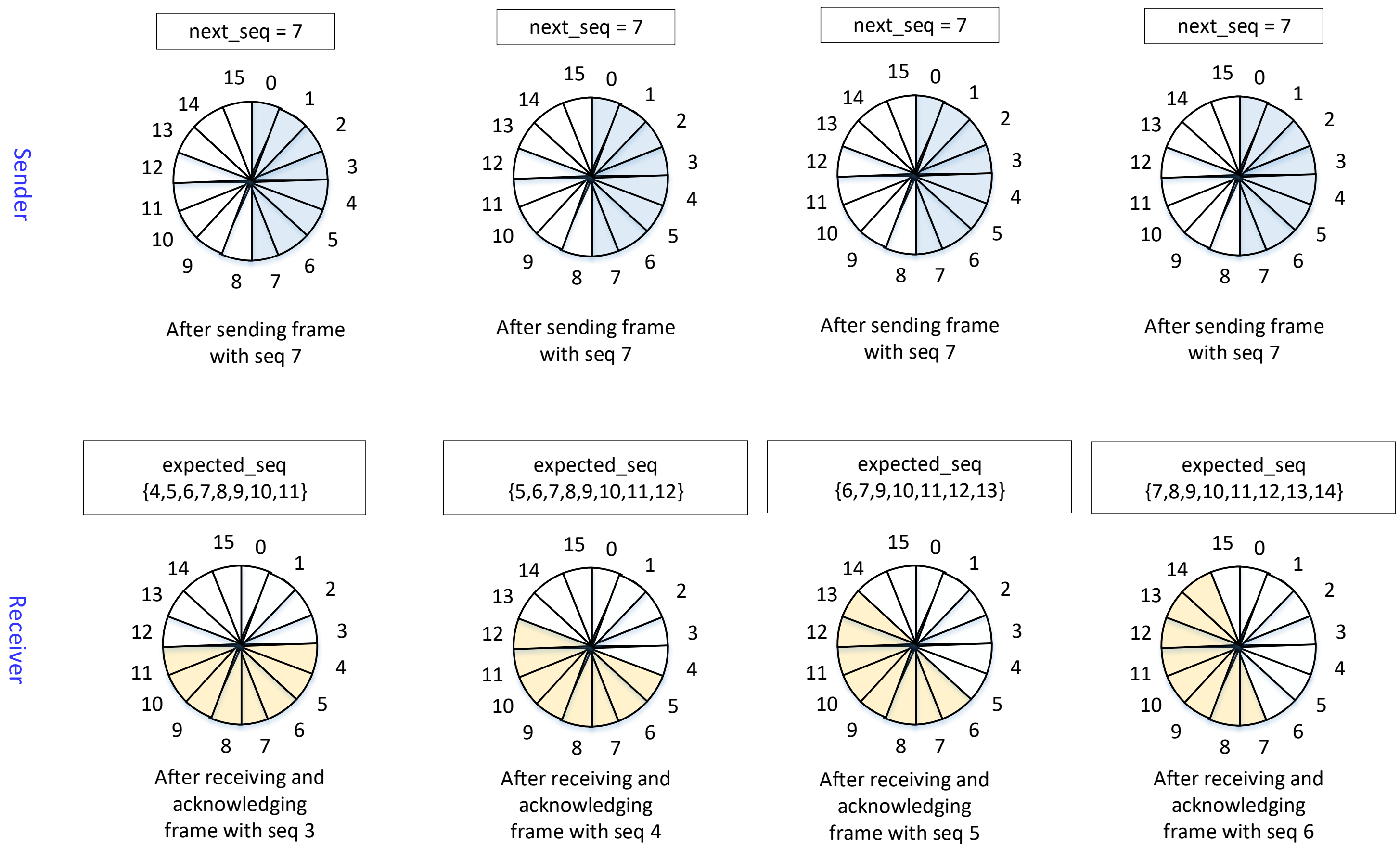
After receiving and acknowledging frame with seq 9



After receiving and acknowledging frame with seq 10

Selective Repeat

Sequence number 4-bit, Window size, Sender 8 Receiver 8



Selective Repeat

- Tradeoff made for Selective Repeat:
  - More complex than Go-Back-N due to buffering at receiver and multiple timers at sender
  - More efficient use of link bandwidth as only lost frames are resent (with low error rates)

- Implemented as p6 (see code in book)

One-Bit Sliding Window

- Transfers data in both directions with stop-and-wait
  - Piggybacks acks on reverse data frames for efficiency
  - Handles transmission errors, flow control, early timers

Each node is sender and receiver (p4):

```
void protocol4 (void) {
  seq_nr next_frame_to_send;
  seq_nr frame_expected;
  frame r, s;
  packet buffer;
  event_type event;

  next_frame_to_send = 0;
  frame_expected = 0;
  from_network_layer(&buffer);
  s.info = buffer;
  s.seq = next_frame_to_send;
  s.ack = 1 – frame_expected;
  to_physical_layer(&s);
  start_timer(s.seq);
```

Prepare first frame

Launch it, and set timer

. . .

One-Bit Sliding Window

Wait for frame or timeout

If a frame with new data then deliver it

If an ack for last send then prepare for next data frame

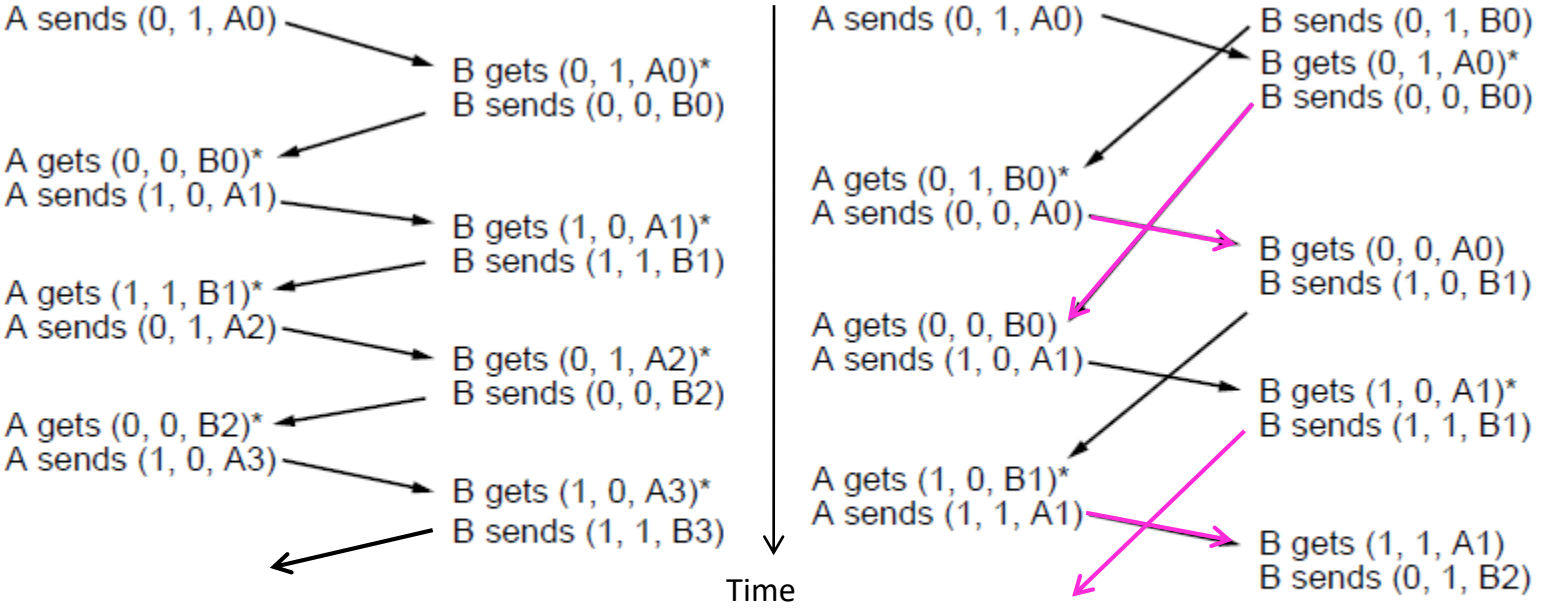
(Otherwise it was a timeout)

Send next data frame or retransmit old one; ack the last data we received

```
. . .
while (true) {
    wait_for_event(&event);
    if (event == frame_arrival) {
        from_physical_layer(&r);
        if (r.seq == frame_expected) {
            to_network_layer(&r.info);
            inc(frame_expected);
        }
        if (r.ack == next_frame_to_send) {
            stop_timer(r.ack);
            from_network_layer(&buffer);
            inc(next_frame_to_send);
        }
    }
    s.info = buffer;
    s.seq = next_frame_to_send;
    s.ack = 1 – frame_expected;
    to_physical_layer(&s);
    start_timer(s.seq);
  }
}
```

One-Bit Sliding Window

- Two scenarios show subtle interactions exist in p4:
  - Simultaneous start [right] causes correct but slow operation compared to normal [left] due to duplicate transmissions.



Notation is (seq, ack, frame number). Asterisk indicates frame accepted by network layer .

Normal case

Correct, but poor performance

Summary

- Connectionless services
- Framing
- Error Control
- Error-Correcting Code
- Error-Detecting Code
- Flow Control
- Data Link Layer Protocols
  - Stop and Wait Protocol
  - Sliding Window Protocol with Go Back N
  - Sliding Window Protocol with Selective Repeat

Next

Medium Access Control Sublayer

- Channel Allocation Problem
- Multiple Access Protocols
  - Pure and Slotted ALOHA
  - Carrier Sense Multiple Access (CSMA)
  - CSMA with Collision Detection (CSMA/CD)
  - Binary Exponential Backoff Algorithm
  - CSMA with Collision Avoidance (CSMA/CA)
- Ethernet and WiFi
- Repeaters, Hubs, Bridges, and Switches