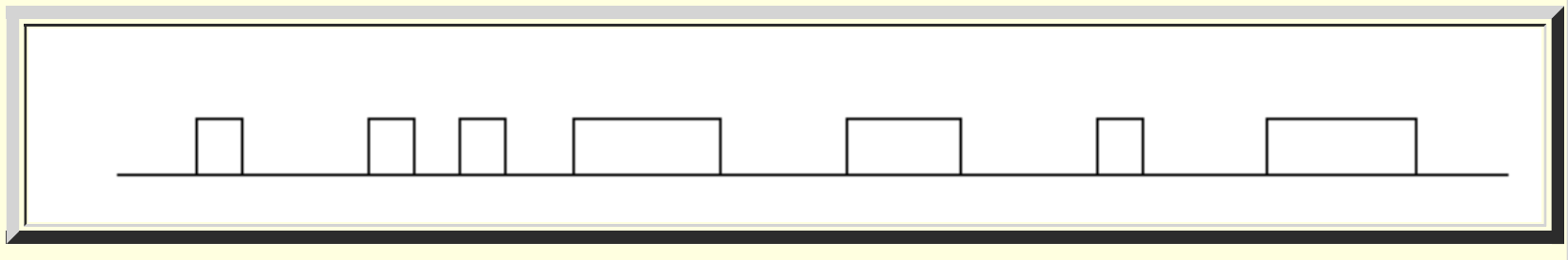
The Datalink Layer

- **Functionality achieved so far...**
  - **Functionality** provided by the **Physical Layer:**

    - **Transmission** of ***analog/digital*** **data** using ***analog/digital*** **signals**

      Example:

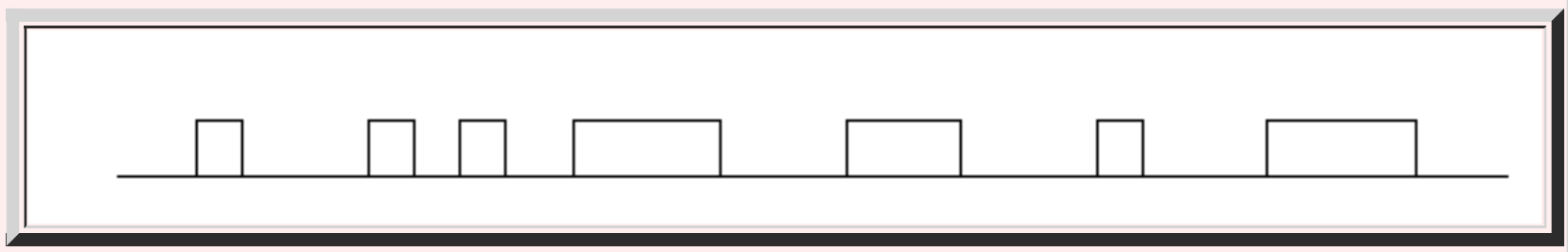
      

    - **Error detection**

- **The framing problem....**
  - **Framing**:

    - We can **transmit** of ***bits*** using ***analog/digital*** **signals**

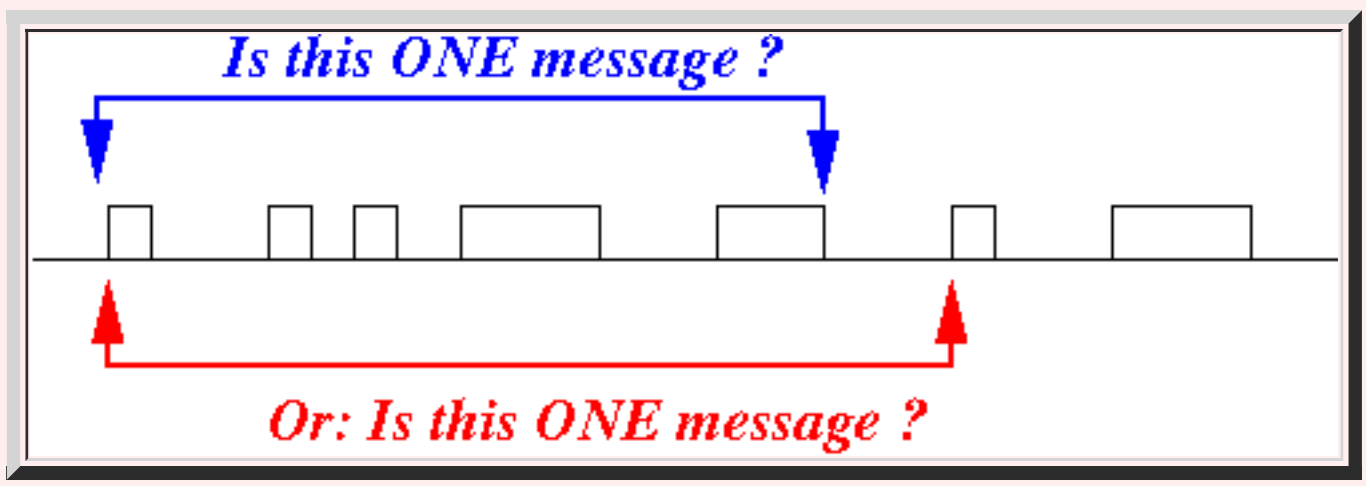
      Example:

      64,000 question:

      - **How** can you ***tell*** the **start** and **end** of a **message** ?

      Example:

      This **problem** is called:

- ***framing***.

- **Terminology: *frame***

  - **Frame:**

    - A **frame** = a ***unit*** **("atom") of data *transmitted*** and ***received*** by the **Data Link layer**

    Therefore:

    - The **Data Link layer** will:

      - **receive *one* entire frame** (if the **frame** is **deemed error-free**) **or**

      - **reject *one* entire frame** (if the **frame** is **deemed** contain **bit errors**)

    - The **Data Link layer** will ***never*** **receive/reject** a ***portion*** of a **frame**

- **Functions provided by the Datalink Layer**

  - **Functions** provided by the **Data Link layer:**

    1. **Framing:**

       - An **protocol (= agreement)** on ***how*** to **group** a **series of bits** into **logical unit (= frame)**

    2. **Error *detection* and *recovery*:**

       1. **Check** if a **received frame** is ***correct***

       2. If **frame** contains **bit errors**:

          - ***Recover*** the **correct frame**

       ***Further* conditions** on **correctness**:

       1. **Each frame must** be **received exactly *once***:

          - ***Duplicate* reception** (that you **cannot tell**) of the ***same* frame** is **in**

fact a communication *error* !!!

2. **Frames** must be ***received*** in the ***same*** **order** as ***transmitted***

3. **Flow control:**

- ***Pace*** the **transmission rate** of the **sender** in accordance with the **processing speed** of the **receiver**

Note:

- The **flow control** **function** is ***desirable*** but is **not** ***necessary*** in the **Data Link layer**

- **Coverage of the Data Link layer**
  - In this **Data Link layer** **section**, we will cover:
    - **Framing**
    - **How to** ensure ***reliable*** **transfer** (**correct** and ***exactly once***)
  - **Flow control:**
    - **Flow control** will **discussed** in the **Transport Layer** **section**

Framing: how to tell when a frame *begin* and when it *ends*

- **Framing**
  - **Try** this **experiment:**
    - Read **these bits** from **left to right**:

      01010001111110000000110100101001001111101111101111011011111101010101

      There **is** a **frame** in this **figure**.....

      Can you tell ***where*** a **frame starts** and **where** it **ends** ???

  - **Fact:**
    - You **cannot** find the **start** and the **end** because:
      - You **do *not* know** the ***protocol* (= agreement)**.....

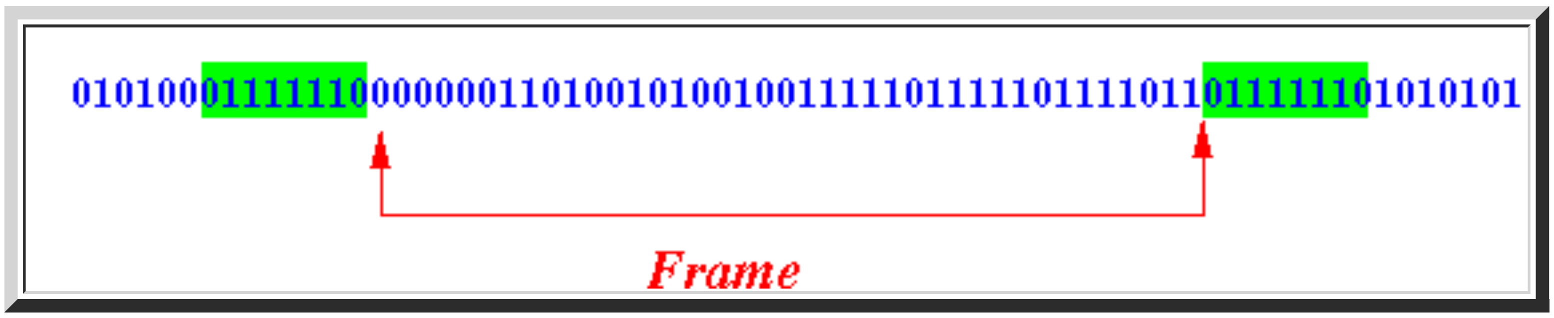
  - ***Further* information:**
    - A **frame *starts*** and ***ends*** with this **pattern**:

      `01111110`

    Can you **now** tell ***where*** a **frame starts** and **where** it **ends**:

    01010001111110000000110100101001001111101111101111011011111101010101

    Answer:



- **Types of framing methods**
  - **Framing techniques:**

    - ***Byte*-oriented** protocols:
      - Used in ***low*** **speed links**
    - ***Bit*-oriented** protocols
      - Used in ***high*** **speed links**

Intro to *Byte*-oriented (framing) protocols

- ***Byte*-oriented framing methods**
  - **Byte-oriented Framing**:

    - ***Byte*-oriented framing** transmits a serie of ***bytes*** in a **frame**

    - A **number** of **byte *patterns* (= code !!)** have a ***special* meaning**

      Example:

      - ***SYNC*** (Synchronize)
      - ***STH*** (Start of header)
      - ***STX*** (Start of text)
      - ***EOT*** (End of text)
      - And so on.

  - **Byte-oriented framing** often **use**:

    - The **ASCII code** to **encode** the **transmitted data**

  - **ASCII table:**

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	@	96	60	`
1	01	Start of heading	33	21	!	65	41	A	97	61	a
2	02	Start of text	34	22	"	66	42	B	98	62	b
3	03	End of text	35	23	#	67	43	C	99	63	c
4	04	End of transmit	36	24	$	68	44	D	100	64	d
5	05	Enquiry	37	25	%	69	45	E	101	65	e
6	06	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	Audible bell	39	27	'	71	47	G	103	67	g
8	08	Backspace	40	28	(	72	48	H	104	68	h
9	09	Horizontal tab	41	29	)	73	49	I	105	69	i
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o
16	10	Data link escape	48	30	0	80	50	P	112	70	p
17	11	Device control 1	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	50	32	2	82	52	R	114	72	r
19	13	Device control 3	51	33	3	83	53	S	115	73	s
20	14	Device control 4	52	34	4	84	54	T	116	74	t
21	15	Neg. acknowledge	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v
23	17	End trans. block	55	37	7	87	57	W	119	77	w
24	18	Cancel	56	38	8	88	58	X	120	78	x
25	19	End of medium	57	39	9	89	59	Y	121	79	y
26	1A	Substitution	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	63	3F	?	95	5F	_	127	7F	□

Notice these **important *codes***:

- **SYN** = **00010110 (16 Hex)**
- **SOH** = **00000001 (01 Hex)** ("start of header")
- **SOT** = **00000010 (02 Hex)** ("start of text")
- **EOT** = **00000011 (03 Hex)** ("end of text")

The *BISYNC* framing protocol

- **Code used in the BISYNC (Byte-oriented) framing protocol**
  - **BISYNC** uses the **ASCII code**:

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	@	96	60	`
1	01	Start of heading	33	21	!	65	41	A	97	61	a
2	02	Start of text	34	22	"	66	42	B	98	62	b
3	03	End of text	35	23	#	67	43	C	99	63	c
4	04	End of transmit	36	24	$	68	44	D	100	64	d
5	05	Enquiry	37	25	%	69	45	E	101	65	e
6	06	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	Audible bell	39	27	'	71	47	G	103	67	g
8	08	Backspace	40	28	(	72	48	H	104	68	h
9	09	Horizontal tab	41	29	)	73	49	I	105	69	i
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o
16	10	Data link escape	48	30	0	80	50	P	112	70	p
17	11	Device control 1	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	50	32	2	82	52	R	114	72	r
19	13	Device control 3	51	33	3	83	53	S	115	73	s
20	14	Device control 4	52	34	4	84	54	T	116	74	t
21	15	Neg. acknowledge	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v
23	17	End trans. block	55	37	7	87	57	W	119	77	w
24	18	Cancel	56	38	8	88	58	X	120	78	x
25	19	End of medium	57	39	9	89	59	Y	121	79	y
26	1A	Substitution	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	63	3F	?	95	5F	_	127	7F	□

- **Idle transmitter in BISYNC**
  - If a **node** that is ***idle*** (= has no frame to send), will **transmit**:

```
0001011000010110000101l0...
SYNC    SYNC    SYNC ....

(00010110 = 16 Hex = Synchronous Idle --- See: ASCII table)
```

  - **Note:**
    - There is ***always*** some ***signal*** **(= voltage)** on a **communication medium (e.g., copper wire)**

- **0 volts** is ***still*** a ***value***
- In fact, **0 volt** is ***low*** and will **convey** ***some*** **meaning**

- **Therefore:**

  - it is ***impossible*** **not** to transmit anything at all !!!

- To **perform** the **"transmit nothing"** function, the **sender** transmits a **serie** of ***null*** **frames**:

```
   00010110000101100010110...

 Interpretation:

   00010110"NULL frame"00010110"NULL frame"00010110...
```

- **Frame transmission protocol in BISYNC**
  - A **node** will **transmit** a **frame** as **follows** (= ***protocol***):

    - **Transmits** the ***Start-of-Header*** **(SOH) code**

    - Then **transmit** the ***header data***

      The **header data** contains:

      - The **sender ID**
      - The **receiver ID**
      - **Other** ***meta*** **data** (such as **priority**, etc)

    - Then **transmit** the ***Start-of Text*** **(SOT) code**

    - Then **transmit** the ***text*** **(= "data")**

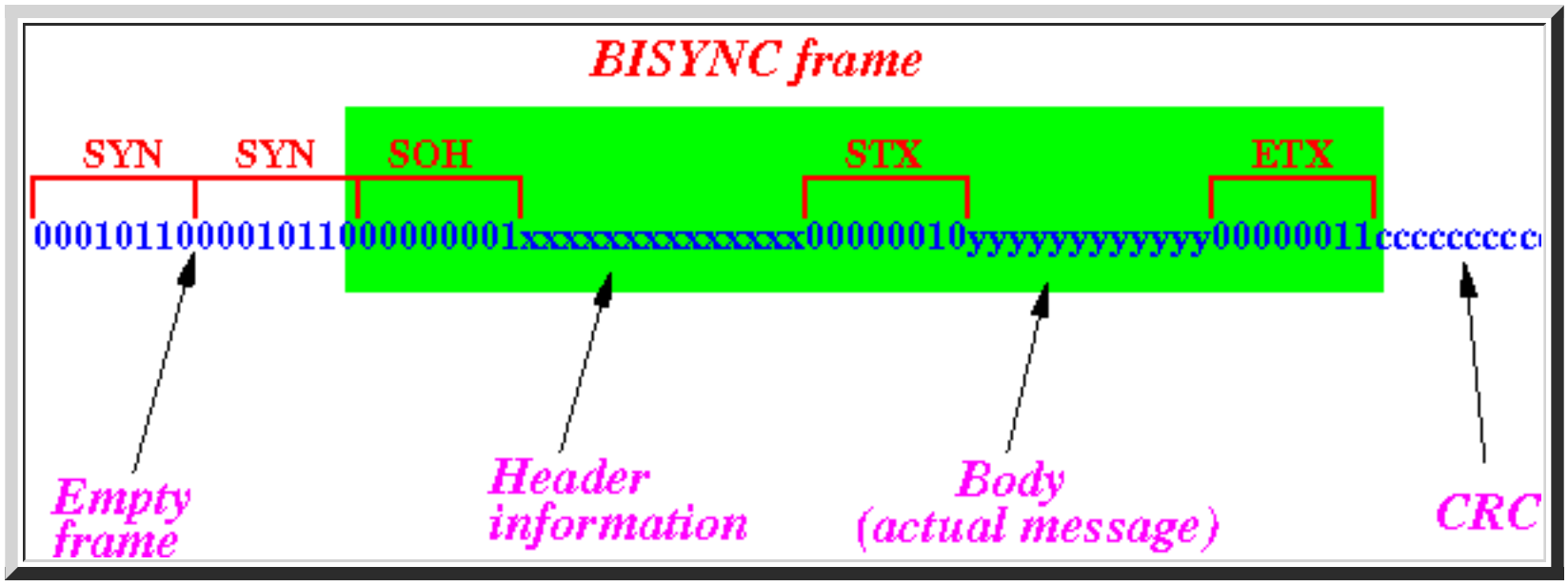
      The **text portion** contains:

      - The **user's data**

    - The **BISYNC frame** is **ended** by the ***End-of Text*** **(EOT) code**

    - A **CRC check sum** is ***appended*** after the **BISYNC frame**

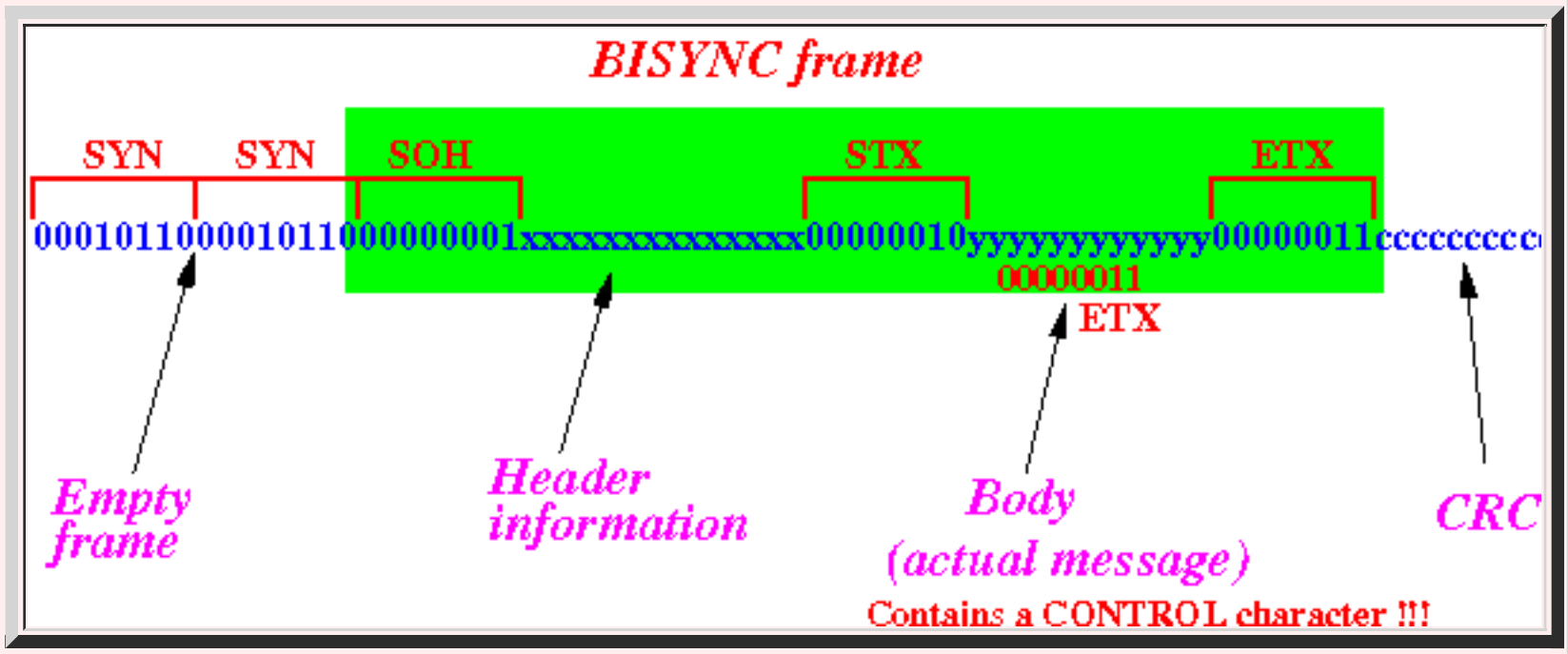
- The **behavior** of a **sender** using **BISYNC** is as **follows**:



- **Transmitting *special* (control) characters: character *stuffing***

  - **Problematic case:**

    - ***What* if** the **header/message** **itself** contains a **special character**, like a ***End-of-Text*** (**00000011**) character:



    This **situation** will **cause** this **error**:

    - The **special character** **EOT** in the **message** will ***terminate*** the **frame** ***prematurely*** !!!

  - **Solution:** use **character stuffing**

    - A ***special* (= control) character** **SPEC** is **transmitted** as **follows**:

DLE	SPEC

    - The **DLE** character is the **Data Link Escape** character and it's **character code** is **00010000**

      - The **DLE** character is **also** a **special character** !!!

- To **transmit** the **DLE character** as **normal data**, we must **transmit** the **following**:

```
DLE DLE
```

The **BISYNC frame** using **character stuffing** has the following **format**:

- The **BISYNC frame** is as **follows**:

```
  SYN      SYN      DLE      SOH
00010110 00010110 00010000 00000001 Header bytes ....

              DLE      STX
.... 00010000 00000010 Message bytes ....

              DLE      ETX         CRC bytes
.... 00010000 00000011 cccccccc cccccccc
```

- **Example:**

- Suppose a **BISYNC frame** contains the **following information**:

```
Header bytes:    00000001 00001111

Message bytes:   00010000 00000011

CRC code:        10101010 10101010
```

- The **sender** will **transmit** the following **BISYNC frame**:

```
  SYN      SYN      DLE      SOH          Header          DLE      STX
00010110 00010110 00010000 00000001 00000001 00001111 00010000 00000010

        Message              DLE      ETX         CRC bytes
00010000 00010000 00000011 00010000 00000011  10101010 10101010
```

Note:

- We **must** use **00010000 (DLE) 00010000 (DLE)** to **transmit** the ***data*** **byte 00010000 (DLE)** !!!

- **Postscript**

  - **NOTE:**

    - **BISYNC** **Wikipedia page:** **click here**

- The **BISYNC protocol** is **no longer used** in **today's networks**... (defined in 1967 !)

The *Point-to-Point* (framing) protocol (PPP)

- **PPP: (point-to-point protocol) Another Byte-oriented framing method**
  - **Some PPP facts:**
    - **PPP** was based on a **very polular data link layer** called **HDLC** (High Level Data Link control)
    - **PPP** was designed to **replace** an older protocol called **SLIP** (Serial Line Internet Protocol)
      - **SLIP** was used to connect **PCs** to the **Internet** at **home**
      - If you use **DSL** at home, then your **Ethernet router** uses **PPP** to communicate with the **DSL modem**.

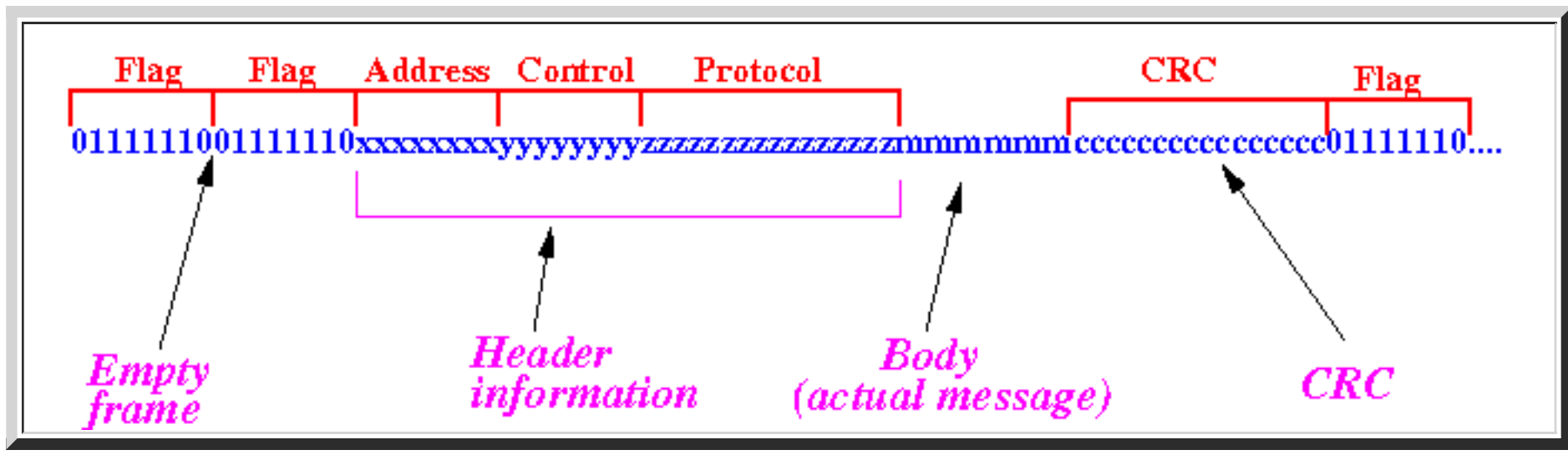
- **Idle transmitter in PPP**
  - If a **node** that is ***idle*** (= has no frame to send), will **transmit** the following ***empty*** **frame**:

```
01111110011111100111111 0....
 Flag    Flag     Flag
```

- **Frame transmission protocol in PPP**
  - A **node** will **transmit** a **frame** as **follows** (= ***protocol***):
    - The **flag** sequence `01111110` (if it has **not** ***already*** transmitted a **flag** previously)
    - A ***one*** **byte address**
    - A ***one*** **byte control code**
    - A ***two*** **bytes protocol code**
    - **User data**
    - **Optional**: one or more ***padding bytes***
    - ***two*** **byte (16 bits) CRC check sum**

- **Flag**

- The **behavior** of a **sender** using **PPP** is as **follows**:



- **NOTE:**

  - The **FLAG (01111110)** character is the ***only*** **special (reserved) character**

  - If the **content** in the ***header*** or ***body*** contains a **FLAG byte**, the **FLAG byte** is **transmitted** as:

    ```
    ESCAPE   Flag
    ```

    (The **ESCAPE** in **data** is transmitted as **ESCAPE ESCAPE** !!!)

- **More info:**

  - Wikipedia: **click here**

Intro to *bit*-oriented (framing) protocols

- **Bit-oriented framing methods**
  - **Characteristics** of ***bit-oriented*** **framing methods:**
    - A ***bit*-oriented protocol** does **not** impart **special meaning** to **byte patterns**

      One **exception:**
      - There is ***one*** **special pattern** to **signal** the ***start*** of a **frame**
      - It's called the ***flag***
    - The ***smallest*** **transmitted unit** is:
      - One **bit** !!!
  - **Note:**
    - **Bit-oriented framing** is **used** transmit **large amount** of **data**
    - **Consequently**:
      - To **avoid** ***clock drift***, a **bit-oriented protocol** use **Manchester encoding**
- **Highlevel Datalink Link Control protocol (HDLC)**
  - **HDLC:**
    - **HDLC** is the **most popular** ***bit*-oriented framing protocol**
      - **HDLC** is a **commonly used Data link layer protocol** on **long haul (wide area) links**
    - **HDLC** is an **ISO standard**
- **The flag pattern of HDLC**
  - **Recall:**
    - **Flag** = a **bit pattern** that **signals**:

- The ***start*** of a **frame** (in a ***bit*-oriented framing protocol**)
- The ***end*** of a **frame** (in a ***bit*-oriented framing protocol**)

- The **flag** pattern of **HDLC** (happens to be a **byte**):

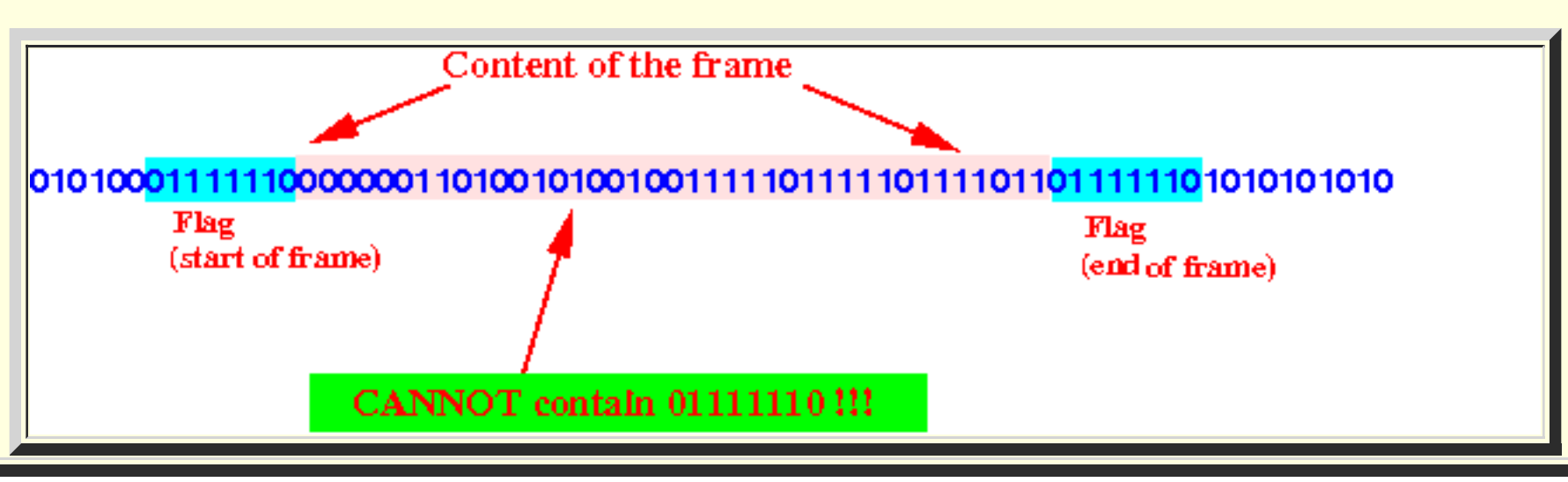
  - **01111110**

  Example:

  - **Find** the **HDLC frame** in the following **transmission**:

    ```
    0101000111111000000011010010100100111110111110111101101111110101010101010
    ```

  - **Answer**:

  - **Remember**: **Frames** are **delimited** by the **flag** sequence which is **01111110**

- ***Problem*:**

  - **What** do you do if the **message** contains the **FLAG (01111110)** ???

  Will be **handled** in the ***next* webpage**.....

- **The idle HDLC transmitter**
  - If a **sender** has **no frames** to send, it **must** transmit the **flag pattern** **continuously**:

    ```
    ....01111110011111100111111O....

       (idle HDLC connection)
    ```

The *bit-stuffing* technique

- **Difference between bit framing and byte framing**
  - **Byte framing**:
    - **Impart *special* meaning** to some ***byte* patterns**

      Example:

      ```
      SOH  (Start of header)
      SOT  (Start of text)
      EOT  (End of text)
      ESC  (Escape !!!)
      ```

  - ***Bit* framing:**
    - Do **not impart *special* meaning** to some ***byte* patterns**

      Therefore:

      - There is **no `ESCAPE`** character in ***bit*-oriented framing protocols** !!!!!

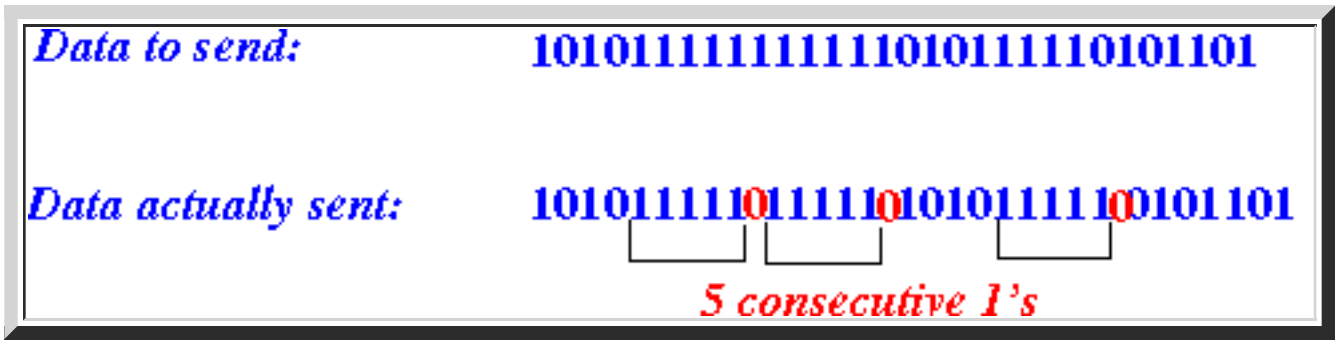
- **Avoid using the flag pattern in a frame**
  - **Fact:**
    - ***Bit*-oriented framing protocols** uses:
      - The ***bit-stuffing* technique**

      to **avoid transmitting** a **flag pattern** in the **data portion**

- **Bit Stuffing**
  - **Bit stuffing transmission rule**:
    - **Whenever** the **sender** has **transmitted 5 *consecutive* "1" bits**:

- the **sender** must ***insert*** **one 0 bit** into the ***transmission***

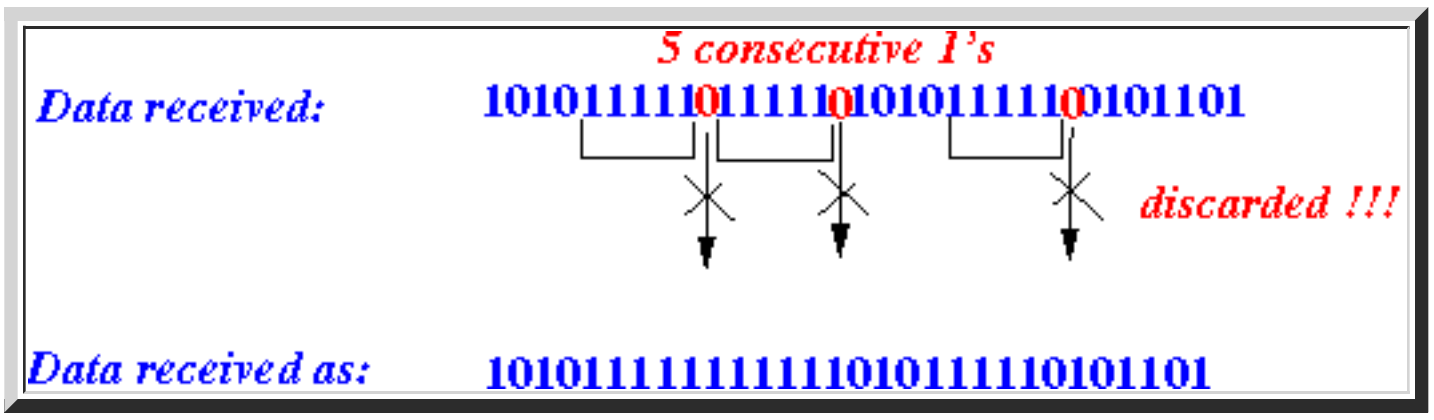
Example:



- **Bit stuffing *receive* rule**:

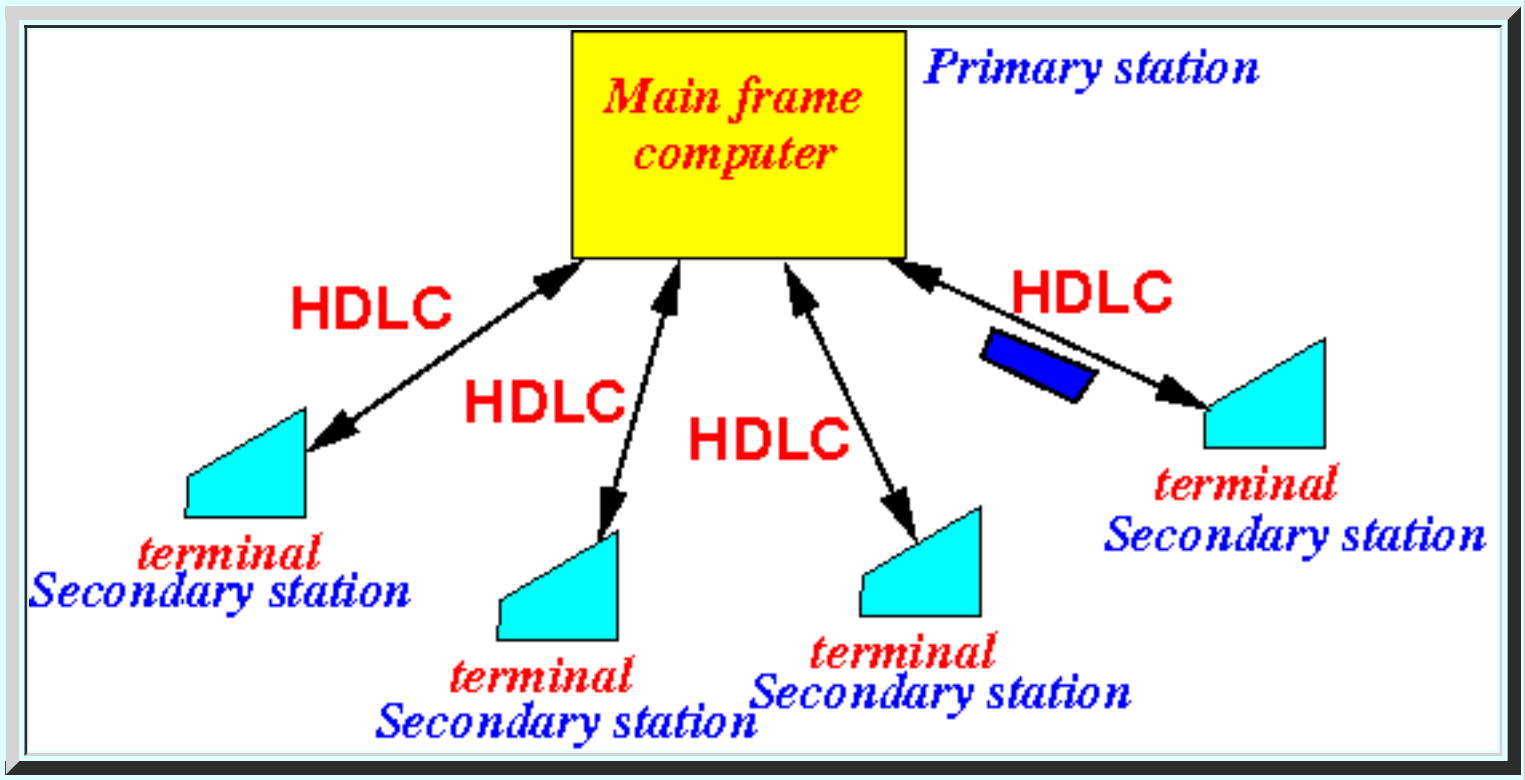
```
When "011111" is received:

   Next bit = 0 ===>  discard this "0" bit (it was a stuffed 0 bit)

   Next bit = 1 ===>  Next bit = 0  ====>  Flag detected
                      Next bit = 1  ====>  Transmission error detected
```

Example:



HDLC: a *popular* bit-oriented (framing) protocol

- **Intro to the HDLC protocol**
  - The **origin** of the **HDLC protocol:**
    - **HDLC** was **originally designed** to connect **one (main frame) computer** to **multiple *peripherals***:



- **Terminology used in HDLC**
  - **Terminology:**
    - ***Primary* station** = an **"*intelligent*" device**
      - A ***Primary* station** is **usually** a **computer**
    - ***Secondary* station** = a ***supporting* device**
      - A ***Secondary* station** is **usually** a ***peripheral* device**
      - **Example**: **termimal**, **printer**, etc.

- **Master-slave protocol**

- **Master-slave protocol:**

  - **Master-slave protocol** = a **model** of **communication** where ***one*** **device** (or **process**) has **control** over the ***other*** **devices**.

  - The ***controlling*** **device** is called: the ***master***

  - The ***other*** **devices** are called ***slaves***

- **Operation** of a **master-slave protocol**:

  - The **master device (= computer)** controls ***every*** **step** of the **communication** !!!

  - A **slave device** can ***only*** **transmit** if the **master device** has given ***permission*** to the **slave device** to do so !!!

- **The original HDLC (datalink) protocol**

  - The ***original*** **HDLC datalink protocol** is **known** as:

    - The ***Normal Response Mode*** **(NRM)**

  - **NRM:**

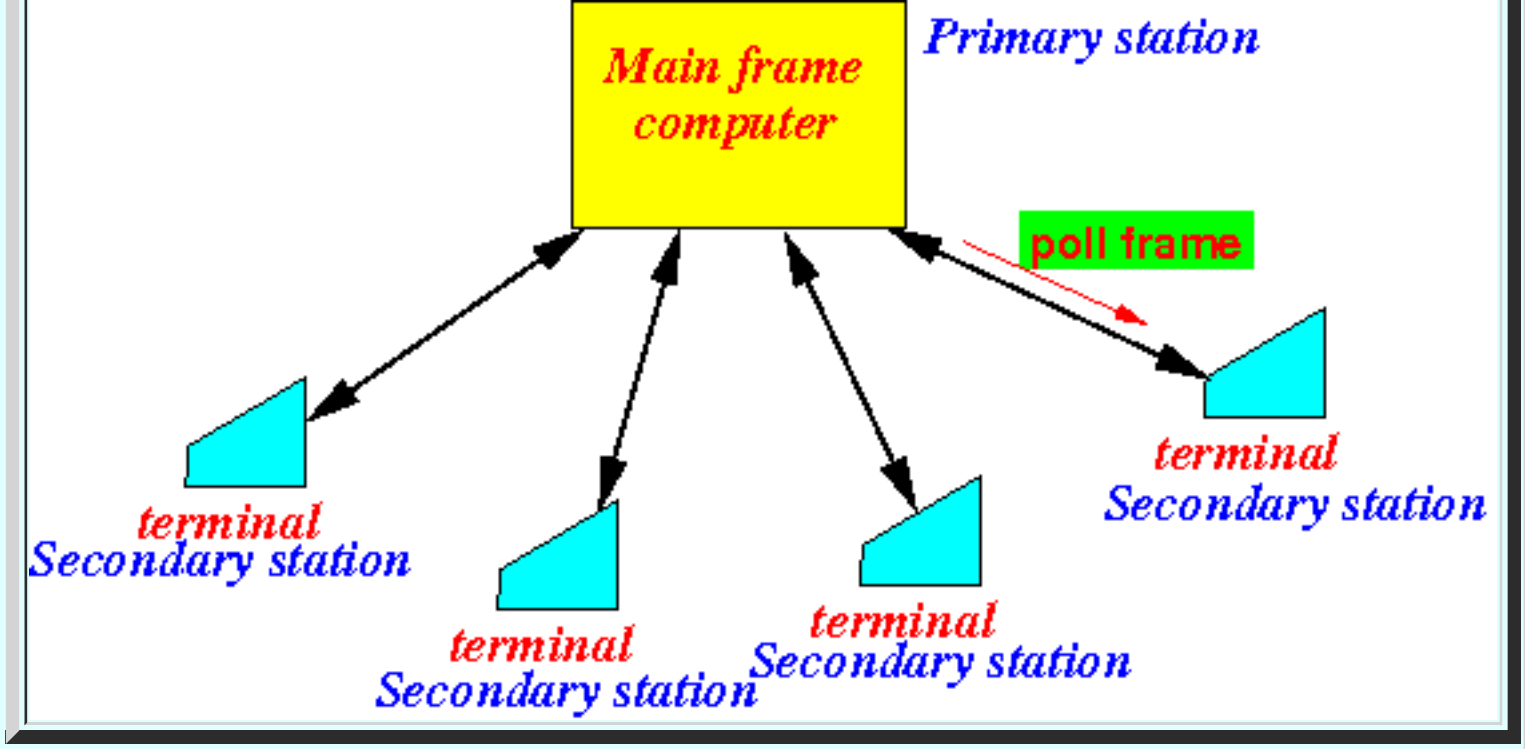
    - The **NRM** of **HDLC** is:

      - A ***master-slave*** **protocol**

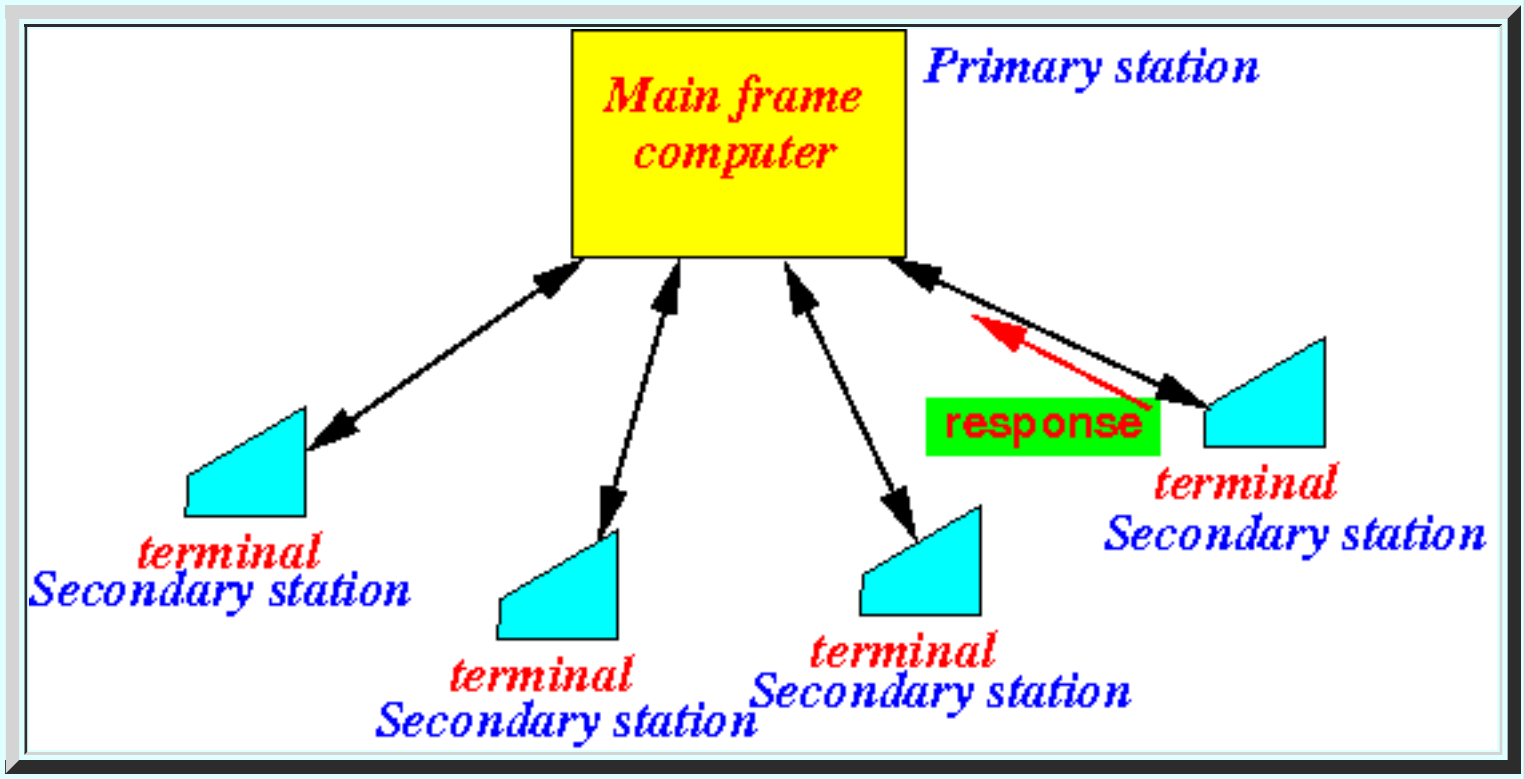
      - The ***primary*** **station** is the **master**
      - The ***secondary*** **stations** are **slaves**

  - **Operation** of the **NRM** protocol:

    - The **primary (= main frame computer)** first **sends** a ***poll*** **frame** to a ***secondary*** **(= terminal)**:



- In **response** to the **poll**, the ***secondary* (= terminal)** will **send** some **data** back to the **primary**:



- **Adapatation for *general* communication**
  - **Later**, the **HDLC** is **updated** with a **new operational mode** called:
    - **Asynchronous *Balanced* Mode (ABM)**
  - The **ABM protocol** is **defined** for:
    - a ***combine* station**

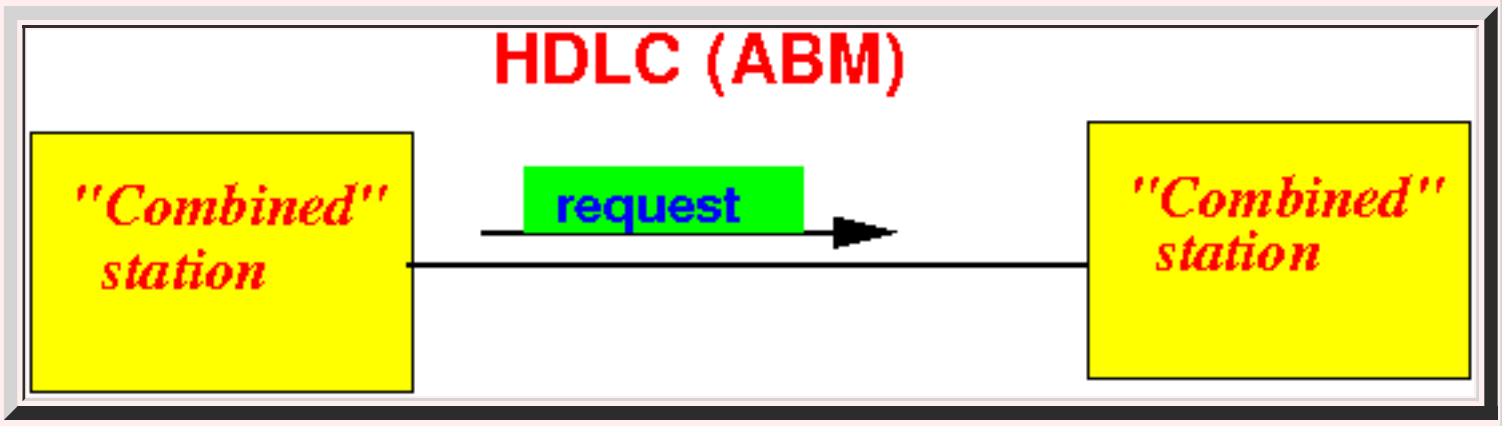
- ***Combined*** **station**:

  - **Combined station** = a **station** that can **operate** as:

    - a ***primary*** **station** (= **takes *control***) and
    - a ***secondary*** **station** (= **process *commands***)

- **Operation** of the **ABM** **protocol**:

  - In the **ABM** mode, ***either*** **device** can **send** a **request** to the **other device**:



  - The **station** that **receives** a **request** will later **send** a **response**:

- **Fact:**

  - The **ABM** **protocol** is a popular **data link protocol** for **long haul links**:

- **Wikipedia page on HDLC**

  - Wikipedia page: **click here**

The HDLC frame *format*

- **Intro to frame formats**
  - **Facts:**

    - **Each *communication* protocol** use it's **own** (**fixed**) **frame format (= structure)**

    - The **frame *format*** is **always** as **follows**:

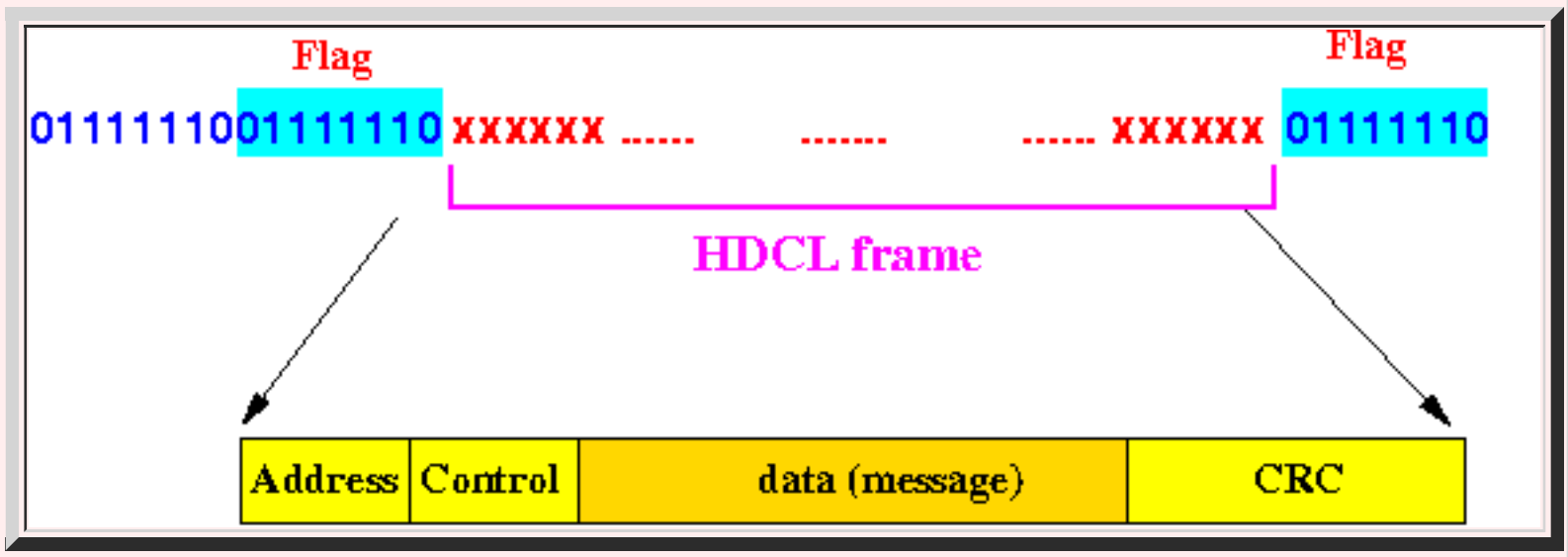
    ```
    +--------------+------------+-----+
    |  header info |  user data | CRC |
    +--------------+------------+-----+
    ```

    - The **header info** contains **information** about:

      - The **identity** of the **sender**
      - The **identity** of the **receiver**
      - The **"identity" (= sequence number)** of the **user data**
      - **etc**, **etc**

- **The structure of an HDLC frame**
  - The **structure** of an **HDLC frame**:

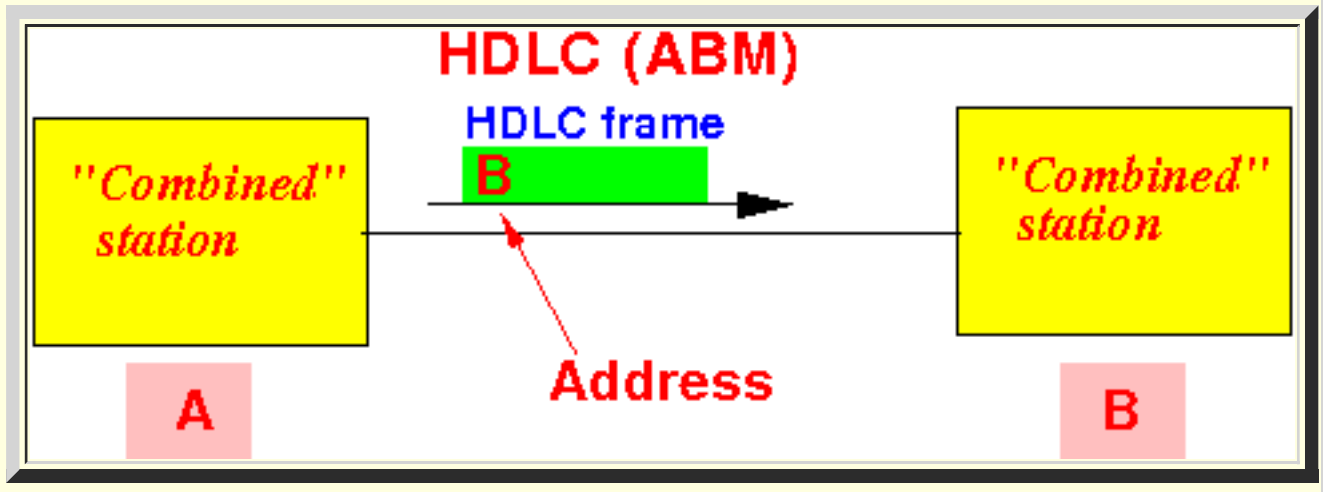
    - The **HDLC frame** is **transmitted** between **2 consecutive FLAG**:



  - The **meaning** of the **fields**:

    1. **Address field**:

- The **address field** contains the **address** of the ***destination*** **station**



---

2. **Control field** contains:

    1. The **frame type**

        - **Identifies** the ***type*** of the **HDLC frame**

    ---

    2. ***Send*** **sequence numbers**

        - **Identifies** the ***position*** of the **frame** in the **sequence** of ***sent*** **messages**

    ---

    3. ***Receive*** **sequence number**

        - Used to ***acknowledge*** the **receiption** of **frames**

        We will discuss the **send** and **receive** ***sequence numbers*** for **error control** soon...

    ---

    4. **Poll indication**

        - **Set** this **bit** to give **permission** to a ***secondary*** **station** to **transmit**

---

3. **Data field**:

    - Contains the **user message**

---

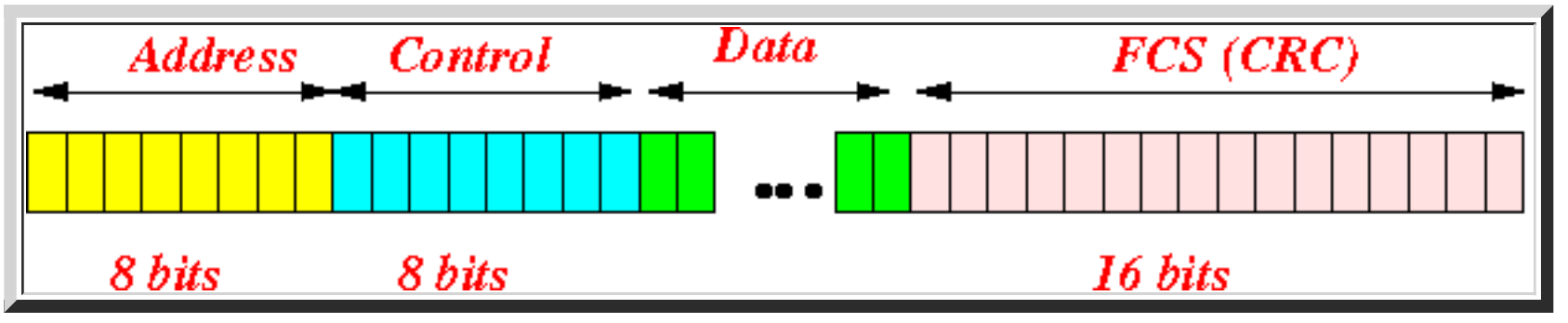
4. **Frame check sequence (FCS)**:

    - Contains a **CRC code** for the ***entire*** **HDLC frame**

- **The length of the HDLC fields**
  - There are **2 version** of **HDLC frames**:

    - **Normal format**
    - **Extended format** (uses **more bits** for **larger values**)

  - The **HDLC frame** in **normal format**:



    The **extended format** uses:

    - **Same** length for **address field**
    - The **control field** is **16 bits**
    - The **FCS field** is **32 bits**

  - **Question:**

    - **How** do we **know** which ***version*** will be ***used*** ???

    Answer:

    - When a **communication device *starts***, it runs an ***initialization*** **protocol**
    - It will **send** **probes** on ***all*** it **communication interfaces**
      - The **device** will **respond** with a **protocol setting message** that includes a **frame type**
    - I.e., the **start up** **protocol** will ***determine*** a **frame type** between the **communicating devices**

The 3 types of HDLC frames

- **Frame types**
  - **Fact:**

    - There are ***different*** **kinds** of **informations** carried in **frames**
    - ***Different*** **kinds** of **informations** will be sent using a ***different*** **type** of **frame**

  - The **3 types** of **frames** used in **HDLC**:

    1. **Information** frame

       - **Information** frames contain **user data**

    2. **Supervisory** frame

       - **Supervisory** frames contains **indications** about whether a **frame** was **received** ***correctly*** or cantains ***bit error(s)***

    3. **"Unnumbered"** frame

       - **Unnumbered** frames contain **link** ***management*** information

- **"Acknowledgement" frame**
  - **ACK frame:**

    - **ACK frame** = a **frame** used to ***aknowledge*** the **correct receiption** of some **data**

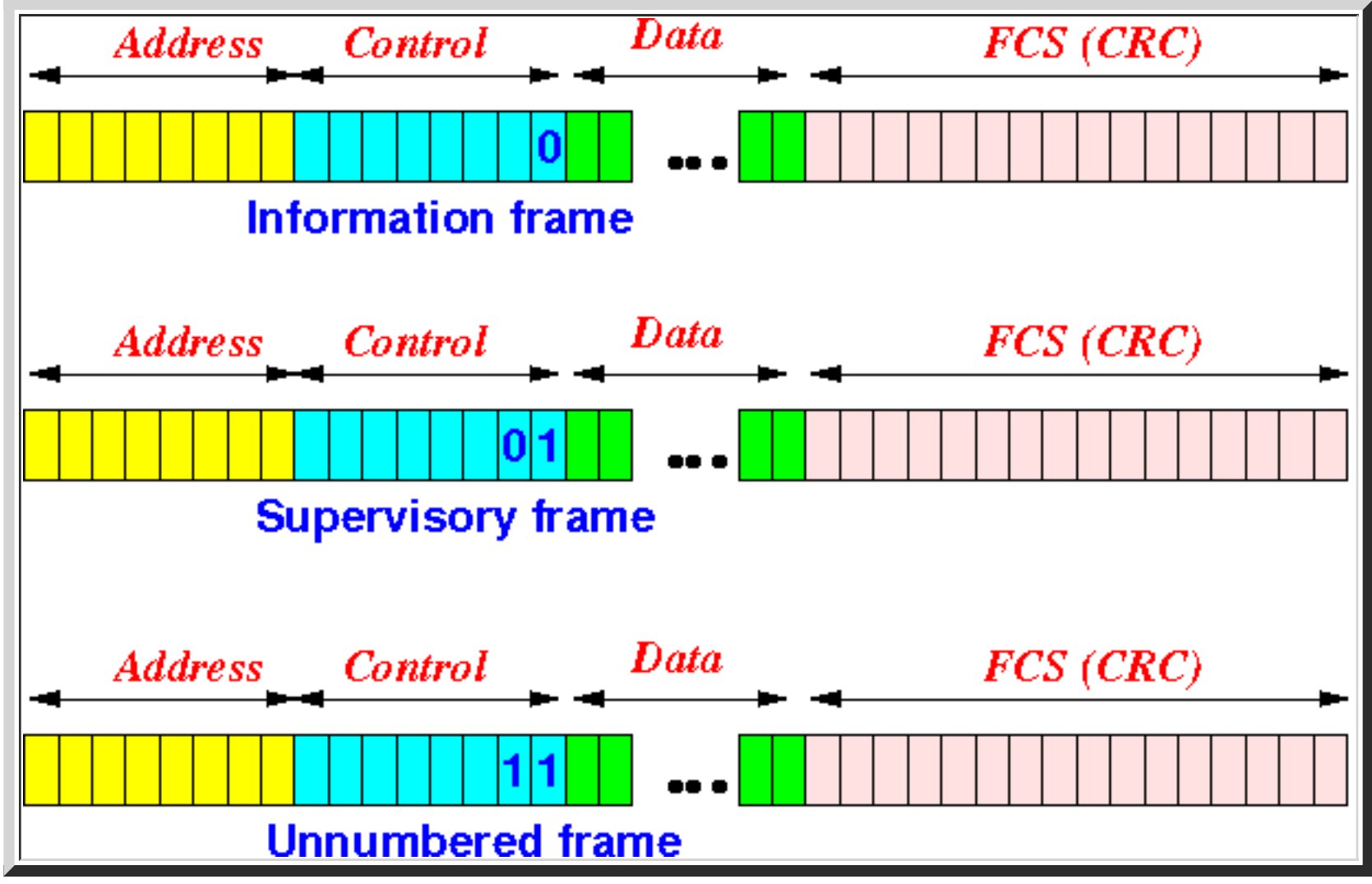
  - ***Positive/negative*** **acknowledgements**:

    - **Possitive acknowledgement** = **indication** that the data was ***correctly*** **received**
    - ***Negative*** **acknowledgement** = **indication** that the data contains ***bit errors*** (and was

discarded)

Communication protocols always use **positive acknowledgements** (because if something is received in error, you can't trust what you has received)

- **How to recognize the type of an HDLC frame**
  - The **value** of some ***control*** **bits** indicates the **frame *type*** as **follows**:

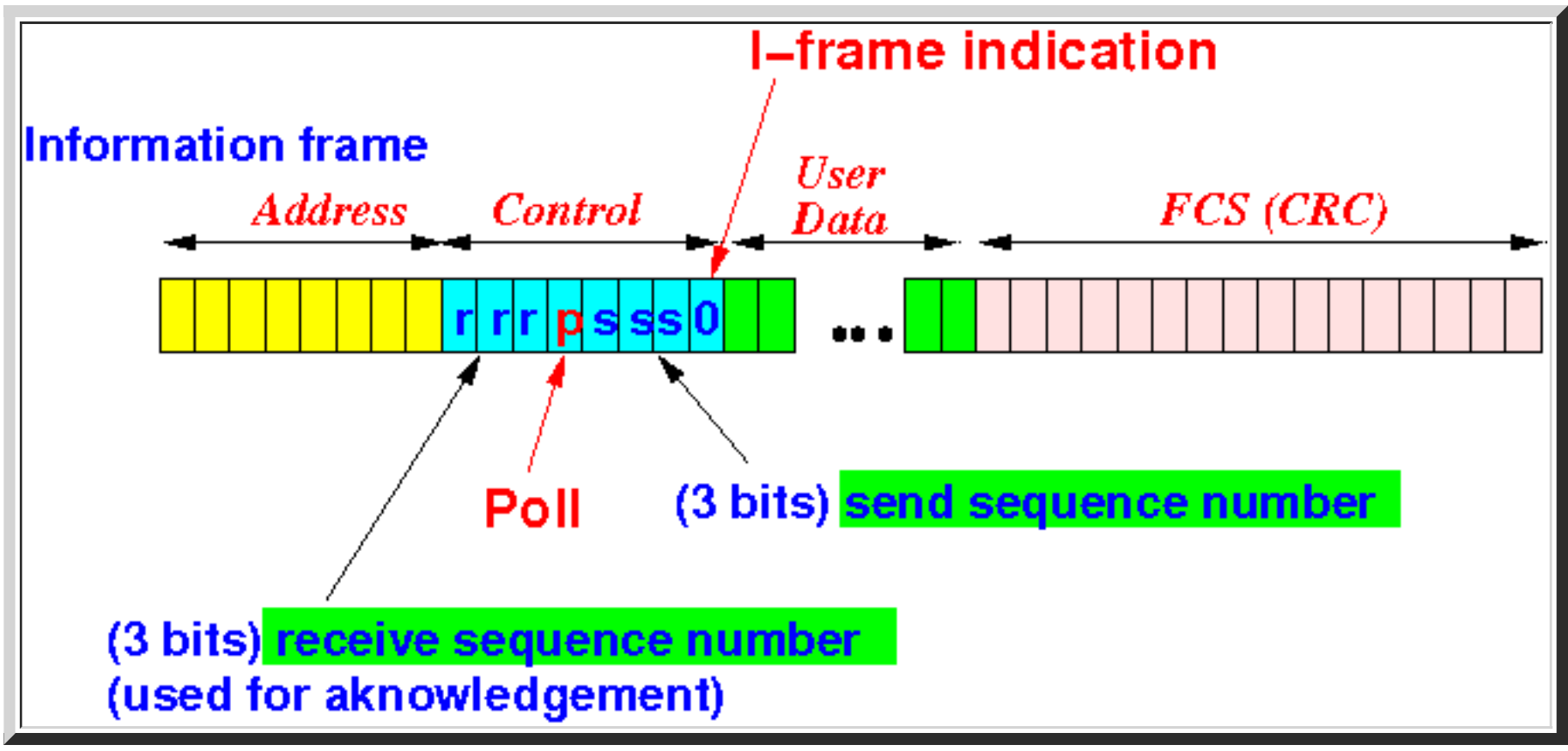


The I-frame

- **The structure of the *information* frame (I-frame)**
  - **Recall:**

    - The **information frame** of **HDLC** contains **user data**

  - **Structure** of the **control field** of an **I-frame**:



Explanation:

- The **bits** `sss` is a **3 bit send sequence number**

  - ***Each* frame** that the **sender** transmits, is identified by a **send sequence number**

    (We will learn **how** this number is used **a bit later**)

- The **bits** `rrr` is a **3 bit *receive* sequence number**

  - The **receive sequence number** is used to **acknowledge** frames that the **sender** has received (from the **receiver**)

    (We will learn **how** this number is used **a bit later**)

- The bit `p` is the **poll** bit

  - When **poll = 1**, the **sender** is **requesting an *immediate* response** from the **receiver**

The S-frame

- **Pre-req to S-frames**
  - **Frame number**:

    - **Frame number** = a **number** used to ***identify***:
      - a ***specific*** **frame** **or**
      - a ***range*** of **frames**

  - **Example: 5** can mean:

    1. the **frame #5**
    2. **All frames** upto **frame #5**

- **The structure of the *supervisory* frame (S-frame)**
  - **Recall:**

    - The **supervisory frame** of **HDLC** contains **positive/negative acknowlegement** for **frames**

  - There are **4 types** of **S-frames**:

    - **Receive Ready *N* (RR)** S-frame:

      - **Acknowledge** the **correct receiption** of **frames** with the given **send sequence number *N***
      - **And:** indicate that the **station** can receive ***more*** **data**

    - **Receive Not Ready *N* (RNR)** S-frame:

      - **Acknowledge** the **correct receiption** of **frames** with the given **send sequence number *N***
      - **And:** indicate that the **station** is ***not*** **ready** to receive ***more*** **data**

        - For example, the **station** is **low** on **buffer space** !!!!

    - **Reject *N* (REJ)** S-frame:

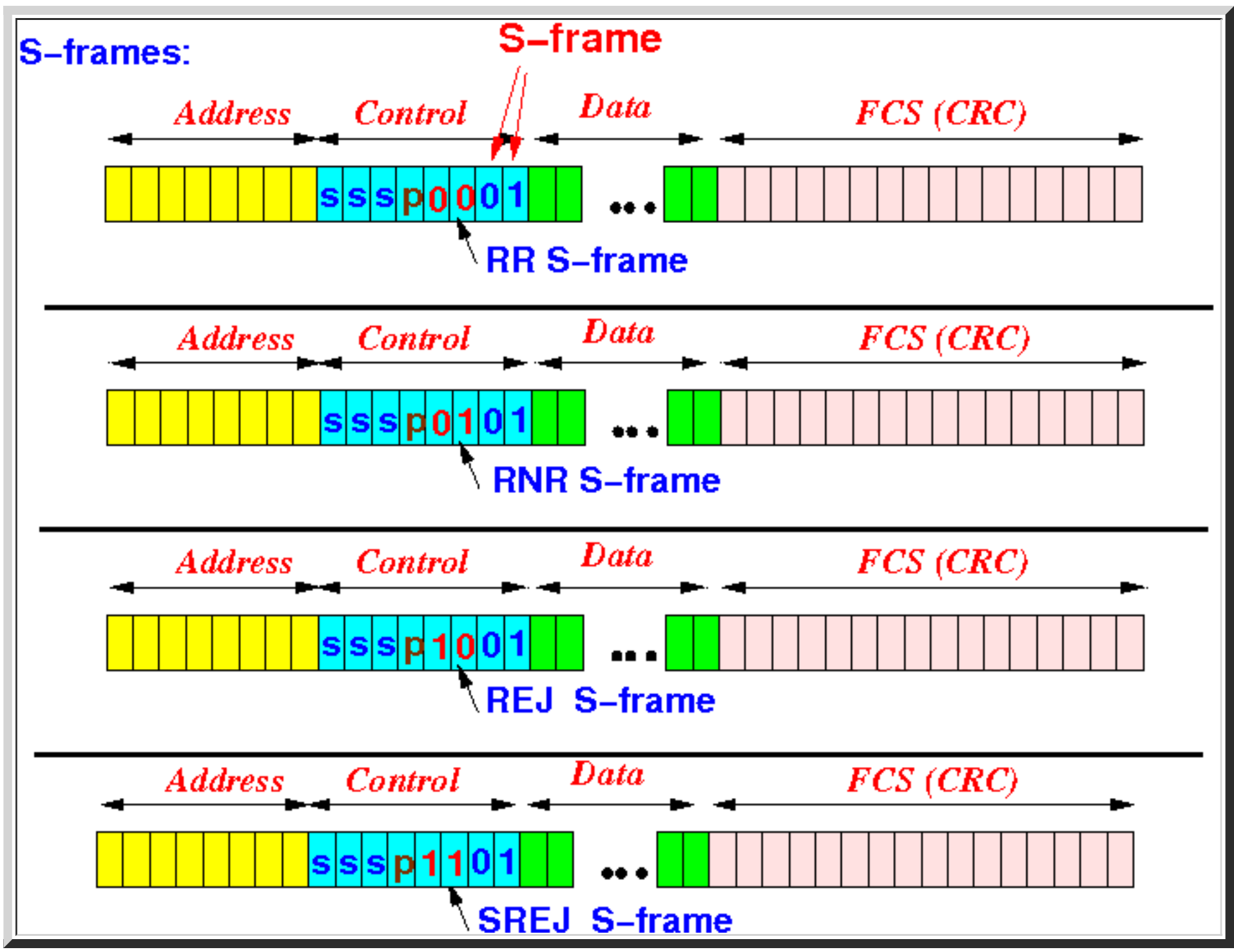
- **Request** the **retransmission** of ***all*** **frames** ***starting*** at the **send sequence number** ***N***

  - **REJ** is a ***negative*** **acknowledgement** !!!

- **Selective Reject (SREJ)** S-frame:

  - **Request** the **retransmission** of the **frame** with the **send sequence number** ***N***

- **How to recognize** the **4 different types** of **S-frames**:



Note:

- The `sss` field **contains** the ***send*** **sequence number** ***N***

  (We will discuss the **use** of this **field** **next**)

Unnumbered frames

- **Unnumbered frames**
  - **Unnumbered frames:**

    - **Unnumbered frames** are used for ***administrative*** **purposes**

      (I.e., **network** ***management***)

    - It is ***not*** used for **network communication**

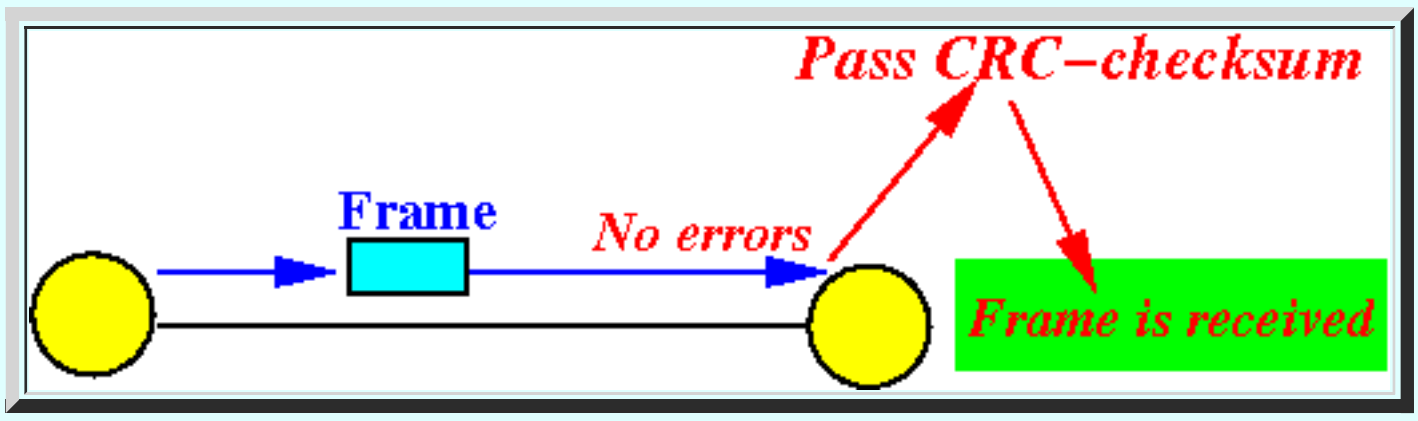
  - **Therefore:**

    - We will **not** concern ourselves with these **frames**

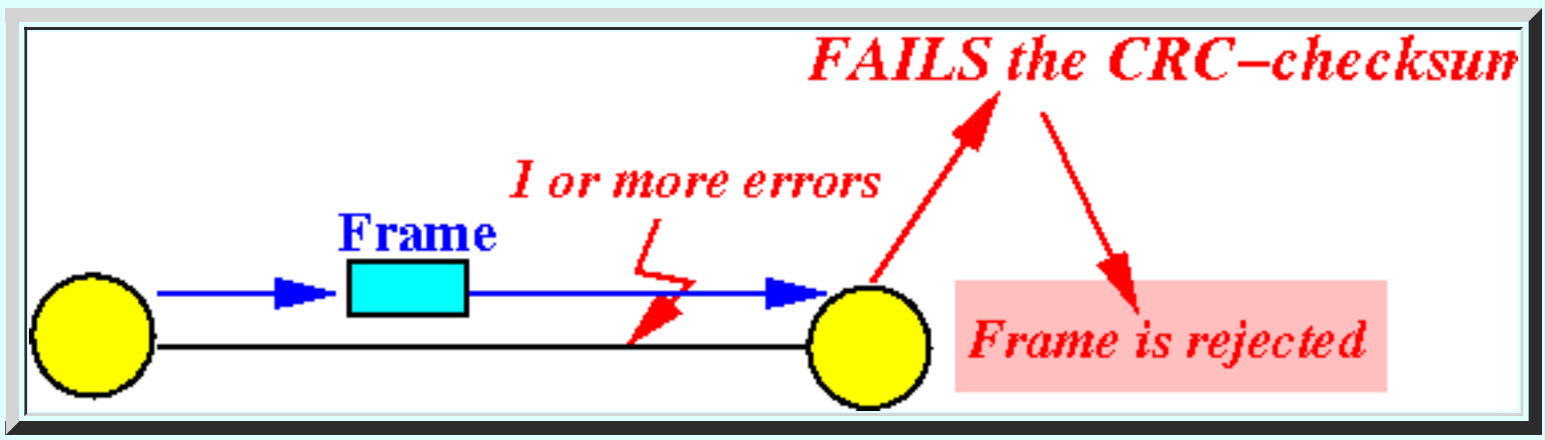
Switched and Broadcast Networks

- **Frame receptions and losses**
  - **How** a **receiver decide** on **reception/loss**:

    - If a ***received*** **frame** passes the ***CRC-checksum*** **test**, the **frame** will be ***received***:



    - If a ***received*** **frame** **fails** the ***CRC-checksum*** **test**, the **frame** will be ***rejected***:



    A ***rejected*** **frame** is ***lost*** !!!

- **The *two (2)* types of Datalink Layers**
  - **Recall:** There are ***2*** **kinds** of **networks**:

    - ***Switched*** **Networks**
    - ***Broadcast*** **Networks**

  - **Properties** of ***switched*** **networks**:

    - **Switched networks** spans ***long*** **distances**
    - **Transmissions** over ***long*** **distance** are ***more*** **susceptable** to ***transmission error***

- Reason: ***noise*** !!!

- The ***main*** **cause** of **frame losses** is:

  - ***bit errors*** (due to **noise**) in the **received frame**

- **Properties** of ***broadcast*** **networks**:

  - **Broadcast network** is ***always*** a ***Local*** **Area Network** that spans ***short*** **distances**

  - **Transmissions** over ***short*** **distance** will ***rarely*** contain ***transmission errors***

  - The ***main*** **cause** of **frame losses** in ***broadcast*** **networks** is:

    - ***collisions*** caused by ***simultaneous*** **frame transmissions** by ***different*** **sources**

- **Consequently:**

  - The **Datalink layer** in **switched networks** will **focus** on:

    - **How to *recover*** a **(lost) frame**

      (to **make sure** the **receiver** will **get** a **frame *correctly***).

  - The **Datalink layer** in a **broadcast network** will **focus** on:

    - **How to *access*** the **transmission medium**

      (to **make sure** there is ***only*** **one transmission** at a time !!!)

- We will discuss the **Datalink layer** of **switched networks** first...

  Then the **Datalink layer** of **broadcast networks**

Introduction: reliable communication

- ***Reliable* communication**
  - A ***series*** of **frames** (or **messages** in general) is **received *reliably*** if and only if:

    1. ***Each* frame (messages)** is received **exactly *once*** by the **receiver**
    2. The **frames (messages)** were **delivered** in the ***same order*** as **transmitted**

- **Terminology: *received* and *delivered***
  - **Received:**
    - A **frame** is **received** = the **frame** has **arrived** at the **receiver *without* errors**

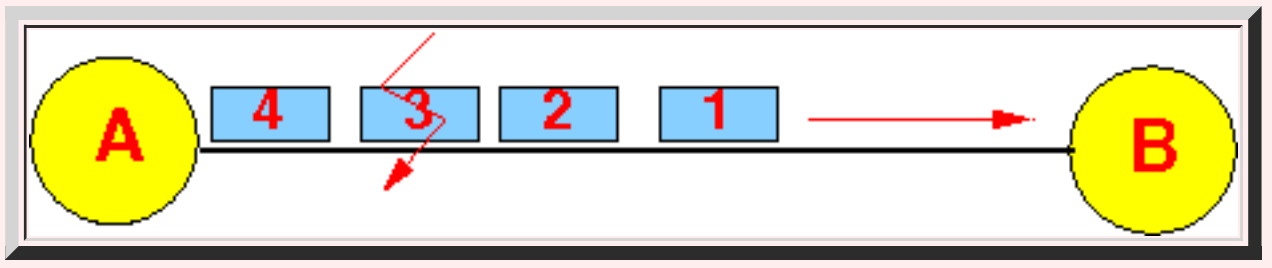
    (**Lost** = ***not* received**)

  - **Delivered**:
    - A (***data***) **frame** is **delivered** = the **(data) frame** has been ***passed* on** to the **user (program)** for **further processing**

  - **Important note**:
    - A **data frame** can be **received** but ***not* delivered**

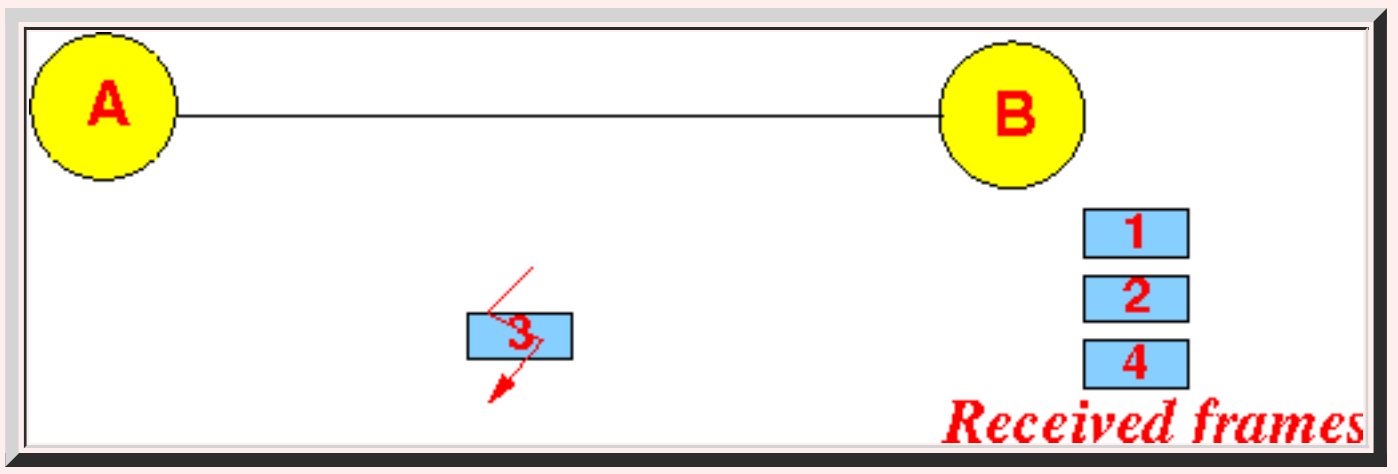
      Example:

      - **A** transmits **4 frames** to **B**:

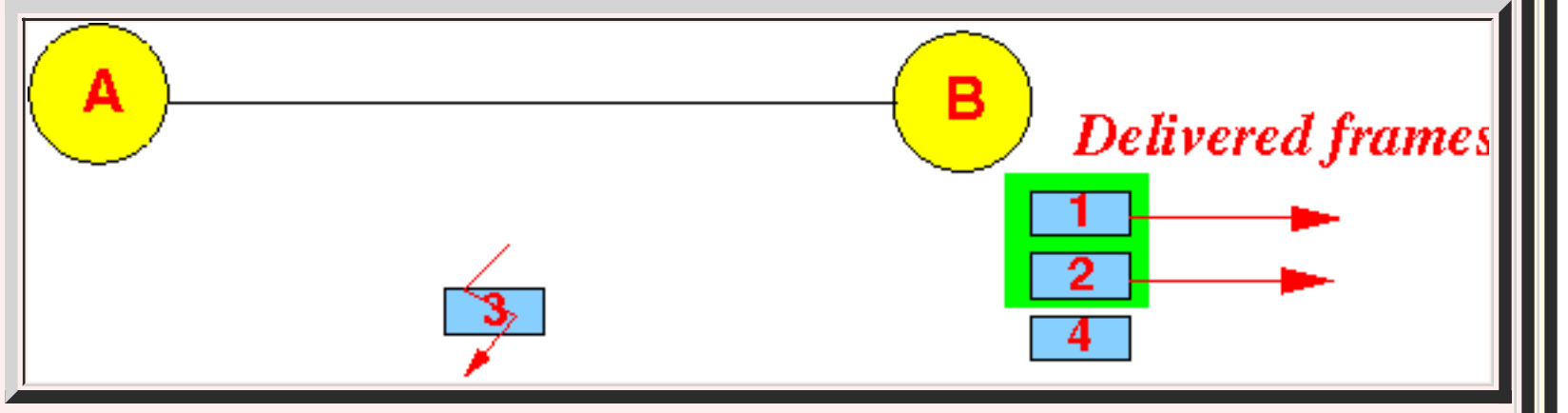


      Frame 3 suffered some ***bit* errors** (and will be **discarded**)

      - **Then: frames 1, 2, 4** have been ***received***:



      - **But only** the **frames 1, 2** can be ***delivered***:



because we must **deliver** the **frame #3** ***before*** we can **deliver** the **frame #4** !!!

- **Error types**
  - **Types** of **errors**:

    - **Frame *loss***:

      - The **frame** was ***not* received** **or**
      - The **frame** was ***received***, **but** contains ***bit errors*** (so it was ***discarded***)

    - ***Duplicate* reception**:

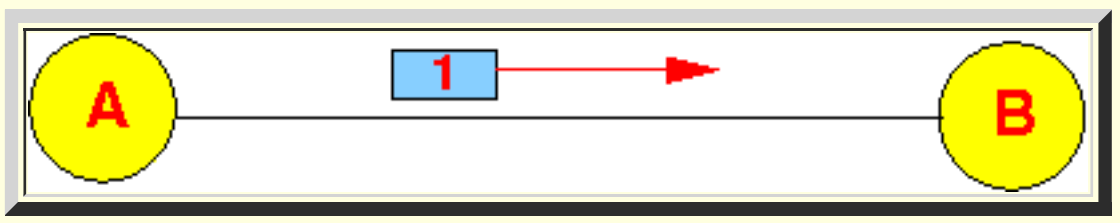
      - The ***same* frame** was **received** ***more* than once**

    - ***Out-of-order* reception**:

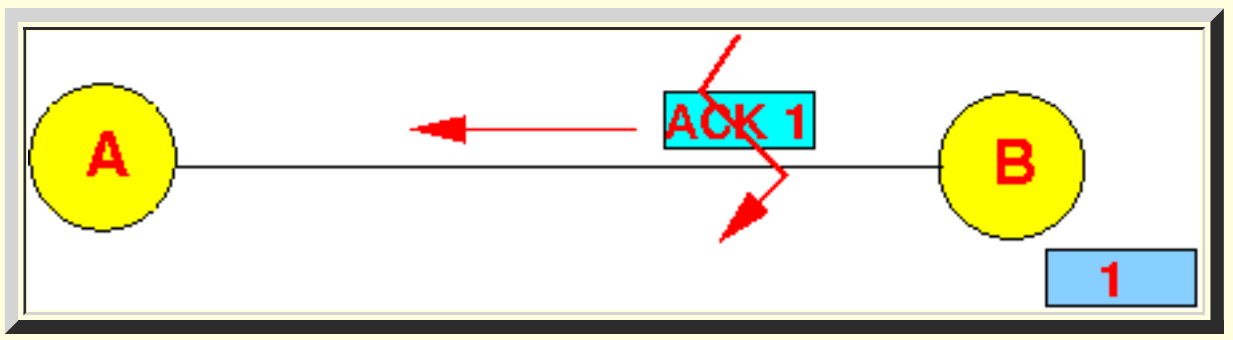
      - **Frames** were **received** in a ***different* order** than the **transmission order**

- **Duplicate frame reception**
  - **Example:**

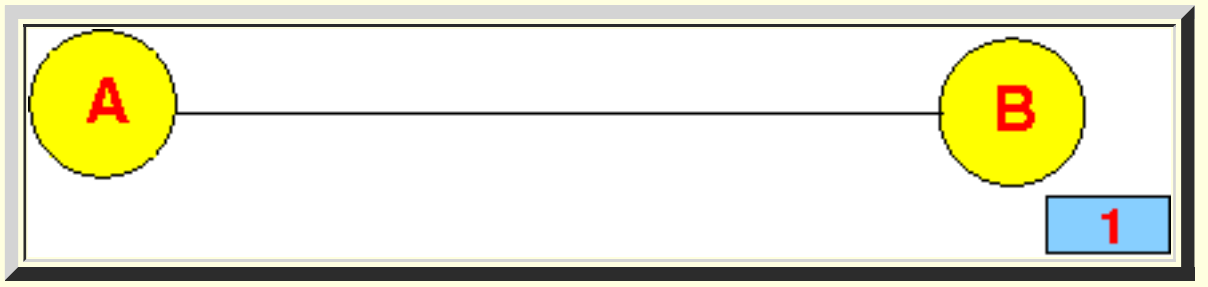
    - **A** transmits **frame 1** and was **received**:



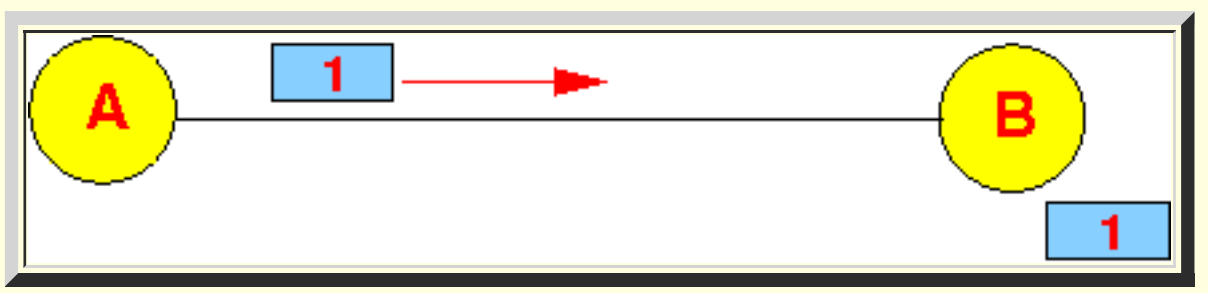
    - **B** sends **back** the **acknowledgement** for **frame 1**, but the **acknowledgement** is ***lost***:



- **A** waits for the **acknowledgement**.... (that is ***not* gonna come**)



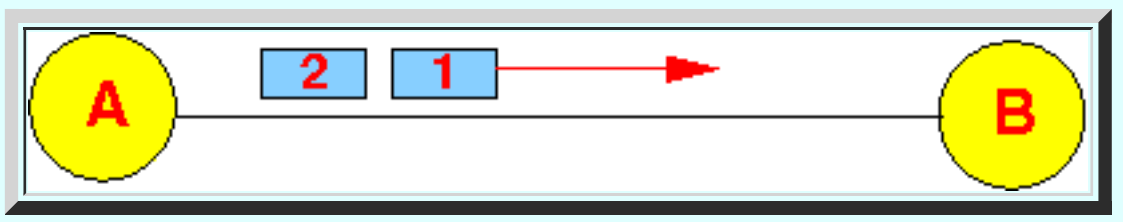
- **A** will the **time out**.... and ***retransmit*** the **frame**:



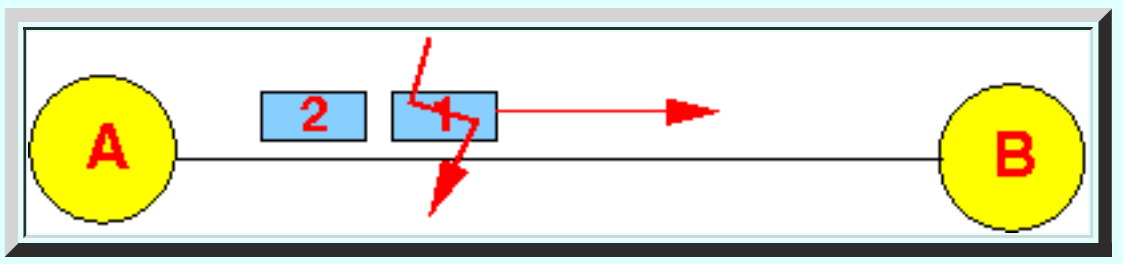
This **frame 1** will be a ***duplicate* frame**

- **Out-of-order receiption on a single transmisiion link**
  - **Example:**

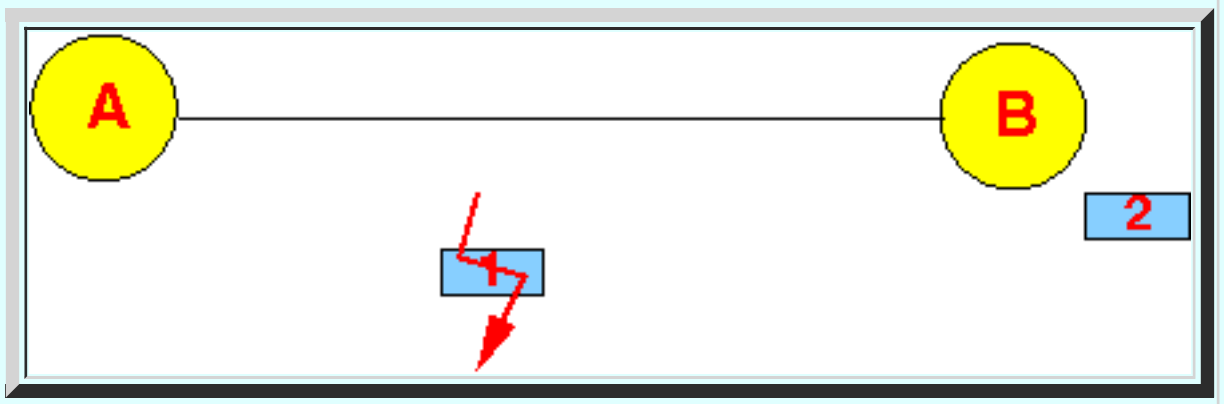
    - **A** transmits **2 frames**:



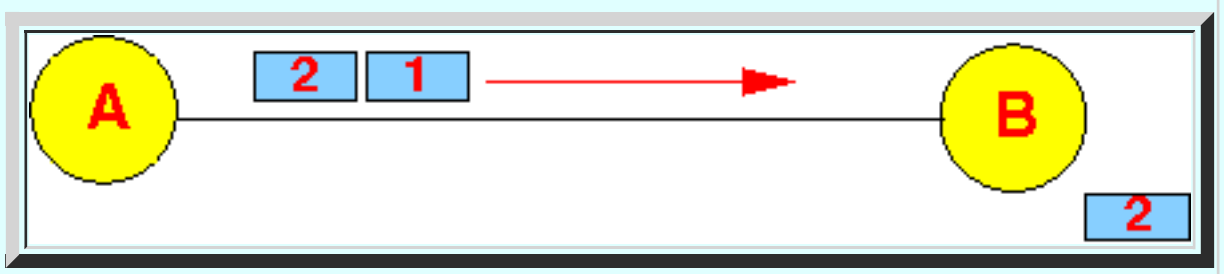
    - **Frame 1** is ***lost***:



    - **B** receives **frame 2** ***first***:



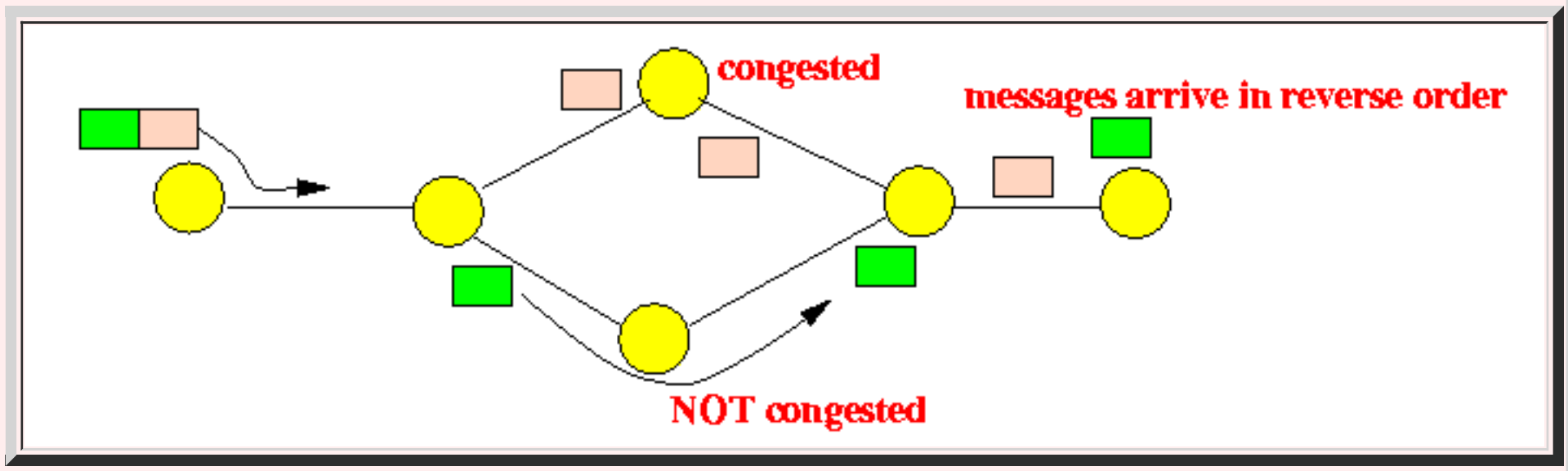
    - **A** times out and **transmits**:



    B will **receive** the **frame 1, 2** in the ***wrong* order**

- **Out-of-order receiption in a general network**
  - **Fact 2:**

    - A **series of frames (messages)** transmitted over ***multiple*** **links** ***can*** be **received** in a ***different*** **order** as **transmitted**:



    (An ***later*** **frame** takes a ***faster*** **route** to the **destination**...

    The ***different*** **frames** ***must*** take ***different*** **routes** for this to happen)

Introduction: the *perspective* of error

- **Protocol**
  - **What** is a **protocol**:

    - **Protocol** = an (**a priori**) ***agreed*** **action and response** between **2 or *multiple* parties** used to **accomplish a task**

  - **Protocol *interactions*:**

    - **Actions** can **cause** the occurence of certain ***events***

      Example:

      - A **frame transmission** (= **action**) will **cause** a ***reception*** **event** at the **receiver**

    - **Events** can **cause** a certain ***response***

      Example:

      - A ***reception*** **event** at the **receiver** can **cause** the **transmission** of an ***acknowledge*** **frame** (= **response**)

- **A warning on the study of protocols**
  - **Communication protocol:**

    - **Communication protocol** = a **set** of **procedure/rules** that **provide** ***reliable*** **communication** between a **sender** and a **receiver**

  - **Warning:**

    - **When** you **study** a **communication protocol**:

      - **You** are **aware** of ***all*** **events** that has **happened**

      In particular:

- **You** are **aware** of the **events** at the ***sender***
- **You** are ***also*** **aware** of the **events** at the ***receiver***

- **In contrast**:

  - The **sender** is ***not*** **aware** of the **events** at the ***receiver***
  - The **receiver** is ***not*** **aware** of the **events** at the ***sender***

- **Warning:**

  - **Make sure** that ***you*** take a **note** of this **fact** when we **study** ***communication*** **protocols**

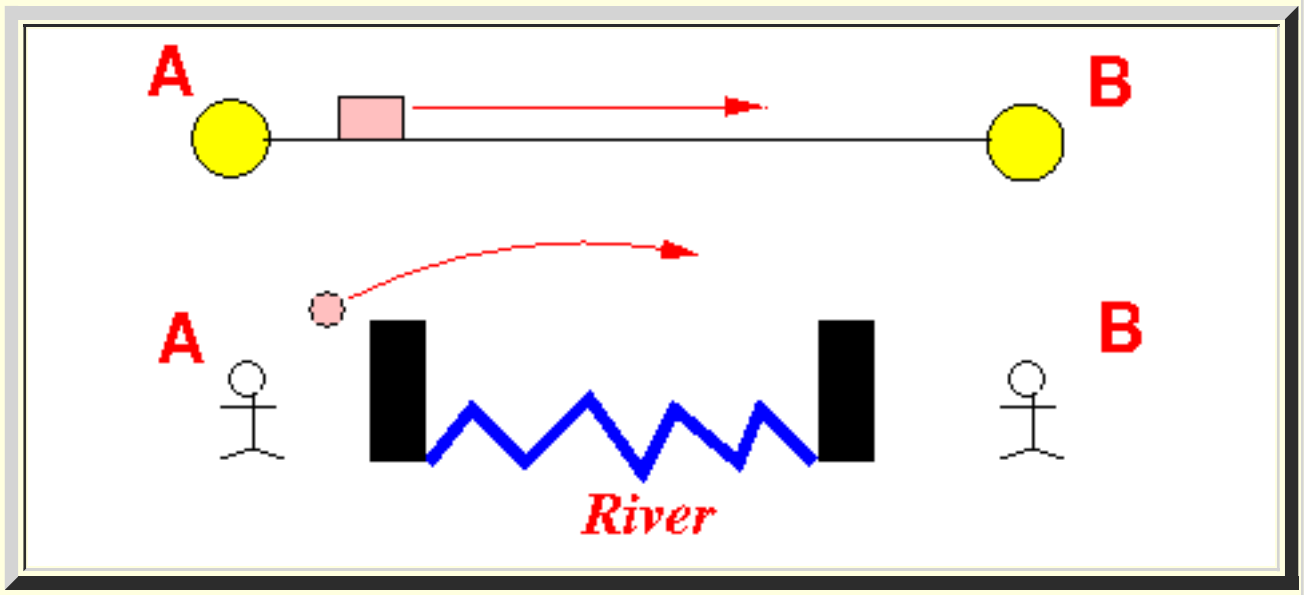
  - **I** call this the ***"omniscient" trap***:

    - **You** have an ***"omniscient" view***
    - **But** the **sender/receiver** themselves do **not** !!!

- **My analogy of computer communication**

  - **Computer communication:**

    - **A** transmits a **frame** to **B**:



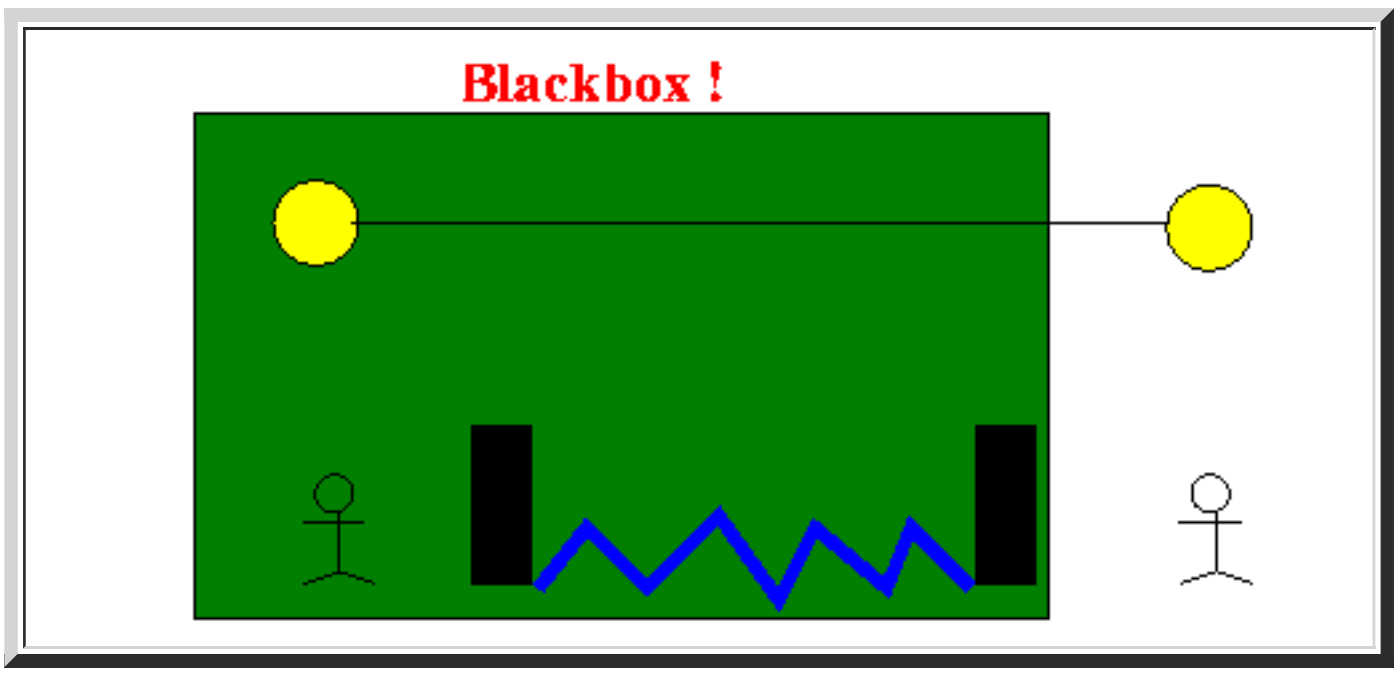
    - **Analogy:**

      - **A** and **B** are **behind** 2 **tall walls**

- **A** and **B** are **separated** by a **river**

- **A** throws **rocks** over the **wall** and **river** to **B**

- **The point of view of the receiver**
  - The **point of view** of the **receiver** is as **follows**:



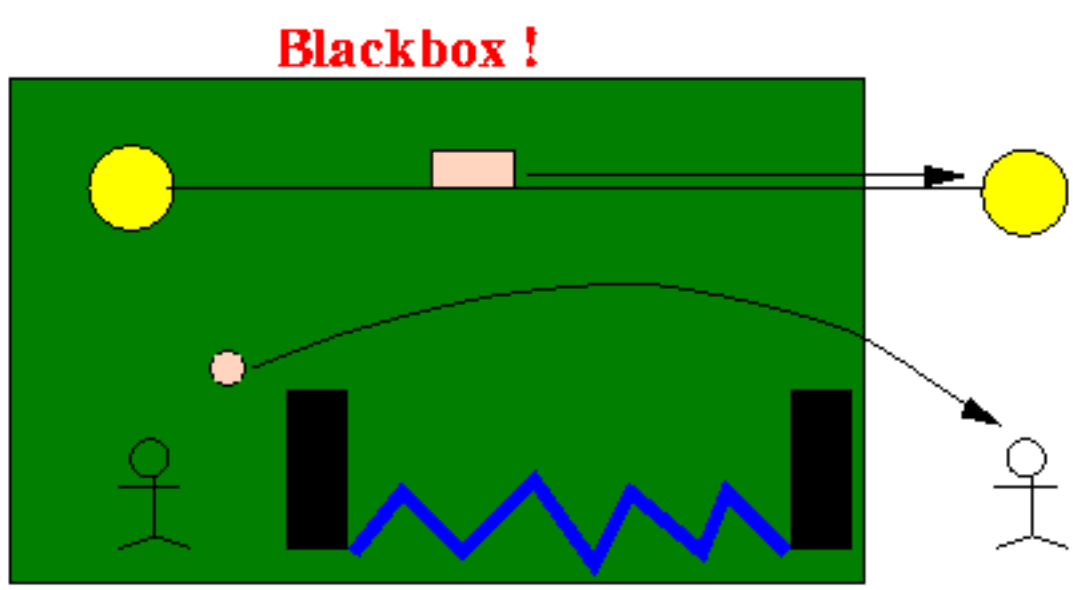
Specifically:

- The **receiver** is ***not*** **aware** of any **event** that has **occured** ***within*** **the** **blackbox** **!!!**
  - **Because** the **wall** prevents the **view** of the **event** !!!

- **Example of the point of view of the receiver**
  - **Example:**

- **Sender** transmits a **frame**

  The **frame** is **received correctly** by the **receiver**:



- This **result** in:

  - **Receiver** receives a **frame**

- In the **analogy**:

  - The **receiver** gets a **rock**

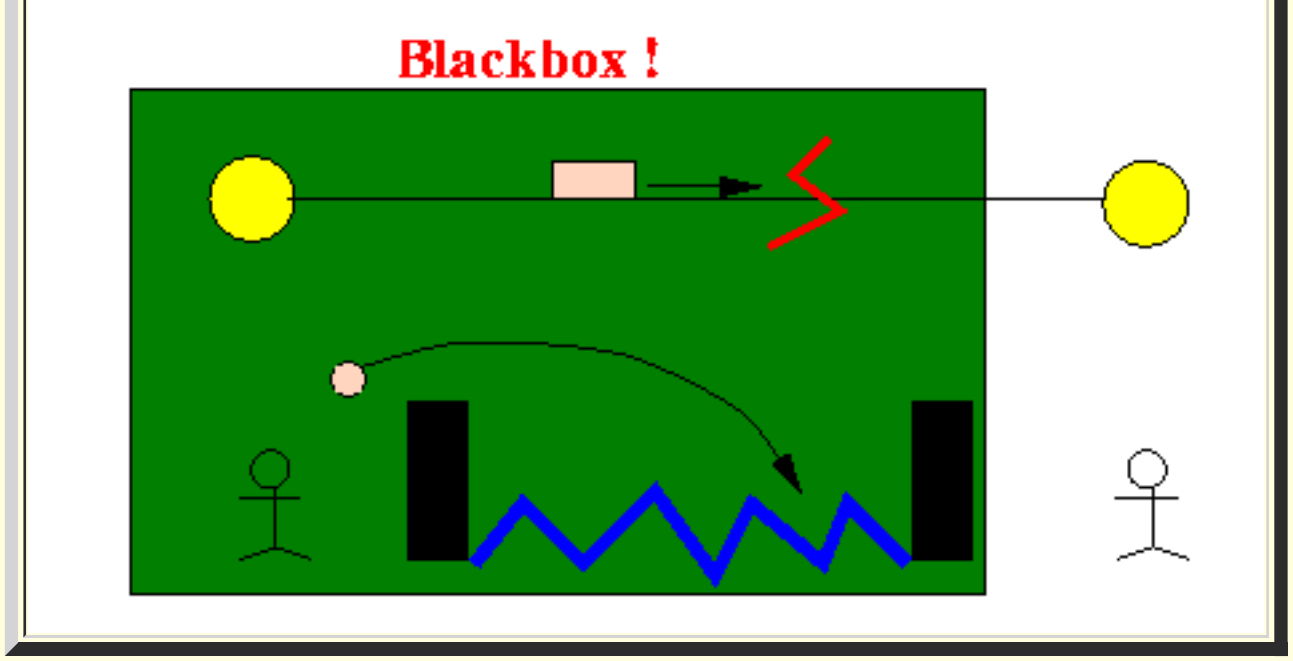
- **Example of the omniscience trap**

  - **Example:**

    - **Scenario 1:**

      - The **sender** transmits a **frame**
      - The **frame** is ***lost*** (e.g., contains ***bit errors***)

      Result:



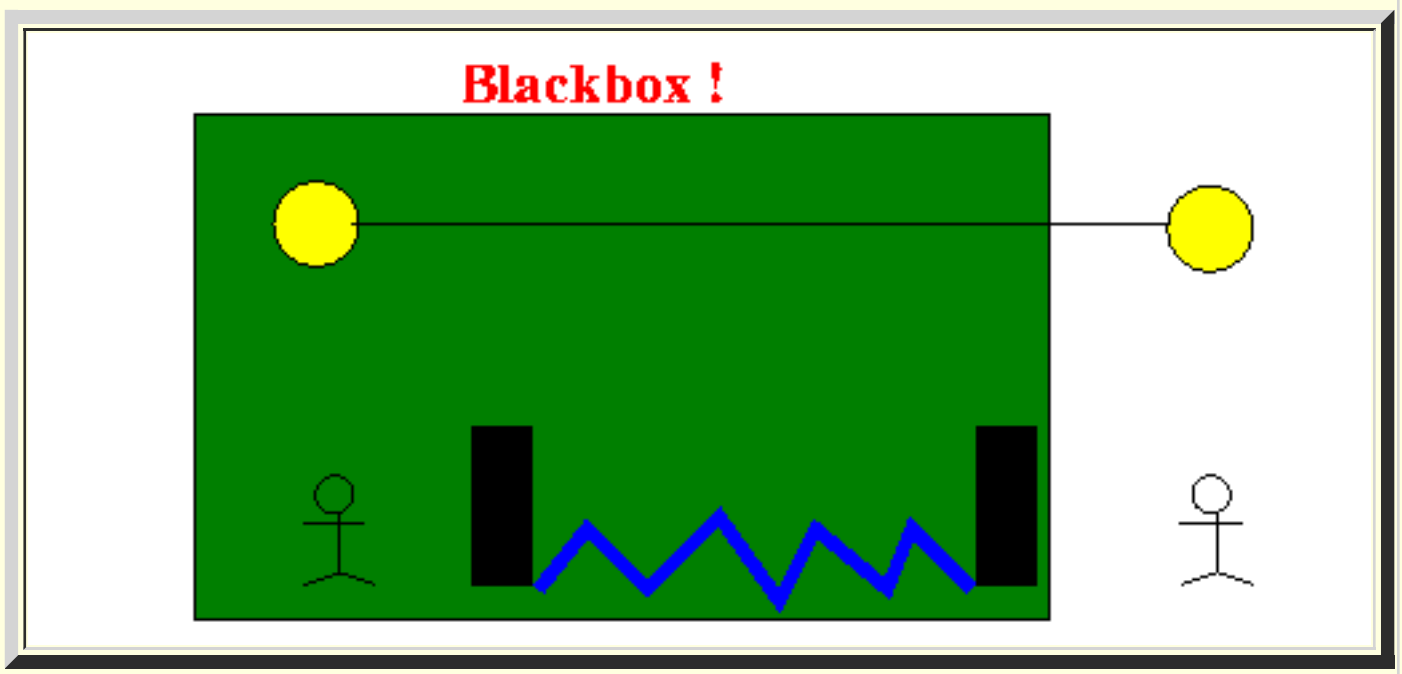
The ***observed*** **event** by the **receiver** is:

- ***No*** **frame** was **received** !!!

- **Scenario 2:**

  - The **sender** did ***not*** perform any **transmissions**

  Result:

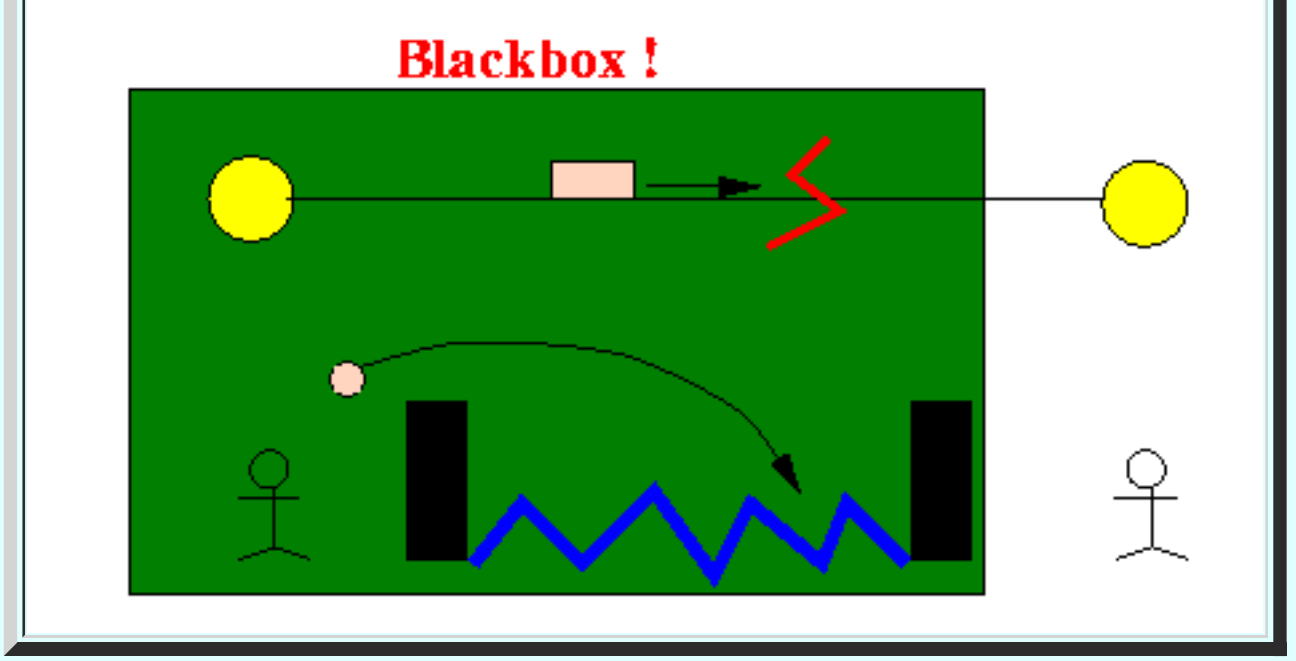


The ***observed*** **event** by the **receiver** is:

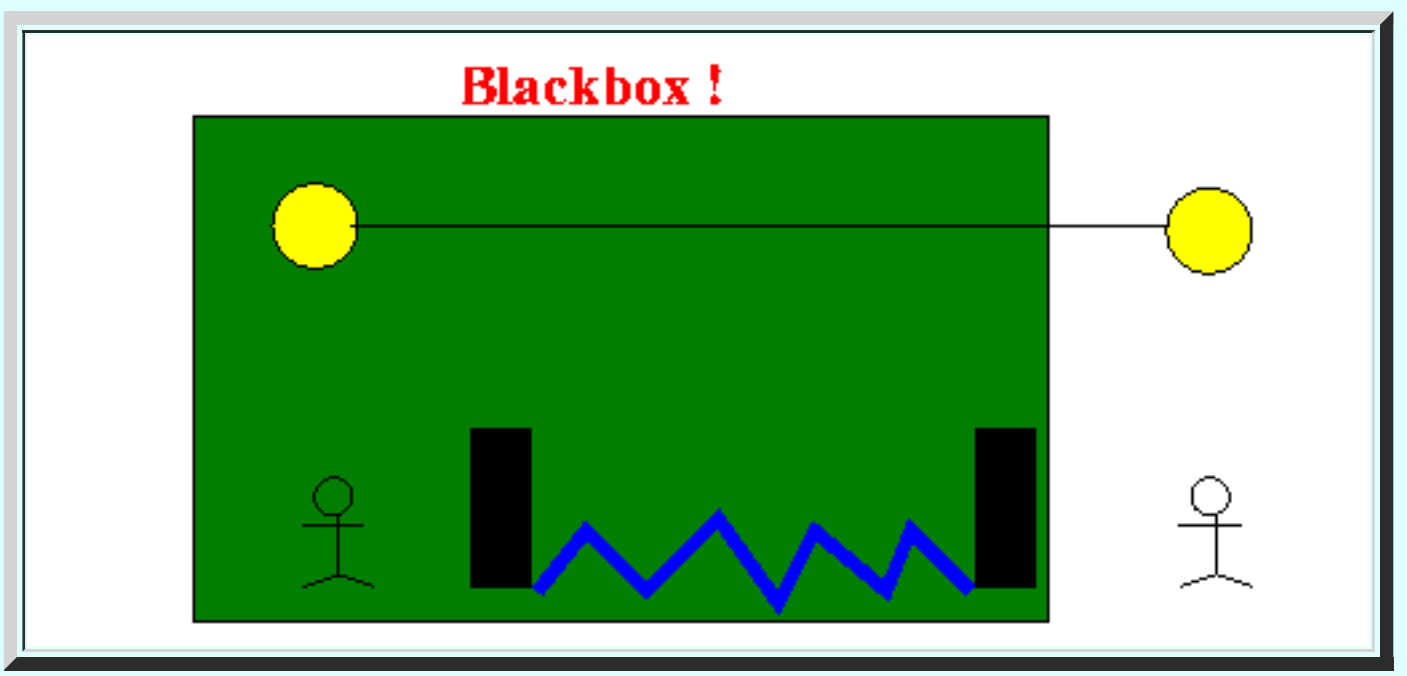
- ***No*** **frame** was **received** !!!

- **Conclusion:**

  - The **receiver** ***cannot*** **tell** the **difference** between **scenario 1**:



and **scenario 2**:



- The **"omniscient" trap**:

  - **Scenario 1** and **2** is ***indistingusihable*** to the **receiver**

  - **However**:

    - ***YOU*** can **tell** the ***difference***

    Warning:

    - To **understand** a **communication protocol**, **you** must **force yourself** to **consider** **Scenario 1** and **2** to be ***identical*** !!!

    (Because the **protocol** must **work correctly** for the **receiver** (and **not** for **you** !!!)

    The **receiver** could **not know** that the **sender** has **transmitted** a frame

!!!)

A *basic* reliable communication protocol (jusk ACK frames)

- **Review**
  - **Recall:**

    - A **frame** received with ***bit errors*** is is **discarded**

    Reason:

    - Since we **do not know *which* part** of the **frame (message)** contains **error(s)**, it would be **foolish** to try to **use** some ***part*** of the **frame**

      So:

      - the **entire frame (message)** is **discarded**

  - **Note that:**

    - **Both**

      - **data frames** **and**
      - **ACK frames**

      can be **lost** (= ***received*** in **error**)

- **The *basic* reliability protocol**
  - Consider the following ***basic* protocol** to achieve **reliable transmission** (that **I** called the ***basic* protocol**):

    - **Sender:**

```
Start:
      Transmit frame;

      Set a timeout;

      while ( not timeout && ACK not received )
      {
         Wait for ACK frame;
      }

      if ( timeout expired )
      {
         go to Start;
      }

      Done; (ACK was received !)
```

    - **Receiver:**

```
      Receive frame;

      if ( CRC check is OK )
      {
         Send ACK frame to sender;

         Deliver frame;
      }
      else
      {
```

```
        Discard frame;
    }
```

- **Protocol analysis**
  - A **communication protocol** must be **correct** under these **circumstances**:
    - **Error-free**
    - With **Errors**

  - **Types** of **errors** you need to **consider** in a **protocol analysis**:
    - **Data frame** is **lost** (bit error(s))
    - ***ACK* frame** is **lost** (bit error(s))
    - ***ACK* frame** is ***late***

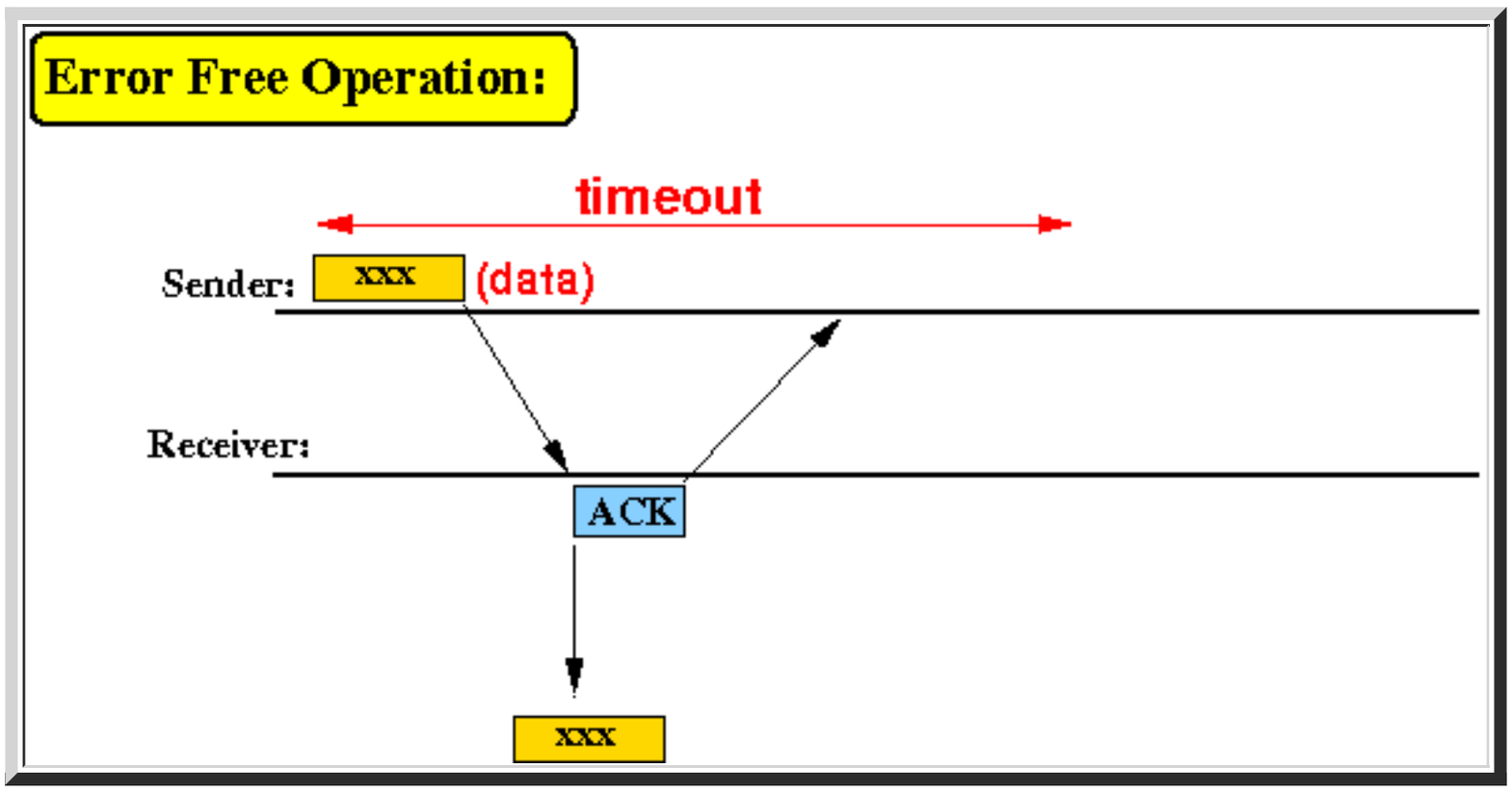
Note:

- The ***duplicate*** **receiption** and ***out-of-order*** **receiption** errors are **caused** by:
  - **Frame loss** !!!

Specifically:

- **Duplicate frame receiption** is **caused** by **lost** of **ACK frame**: **click here**
- **Out-of-order frame receiption** is **caused** by **lost** of **data frame**: **click here**

- **The *basic* protocol in error-free operation**
  - **Example**: ***error-free*** operation



According to the ***basic*** **protocol**:

1. **Sender** transmits **data frame** containing **xxx**
   - Sender **sets** a **timeout** for **frame**

---

2. **Receiver** receives the **data frame**

   - **Receiver** transmits a **ACK frame**
   - **Receiver** delivers the **data frame**

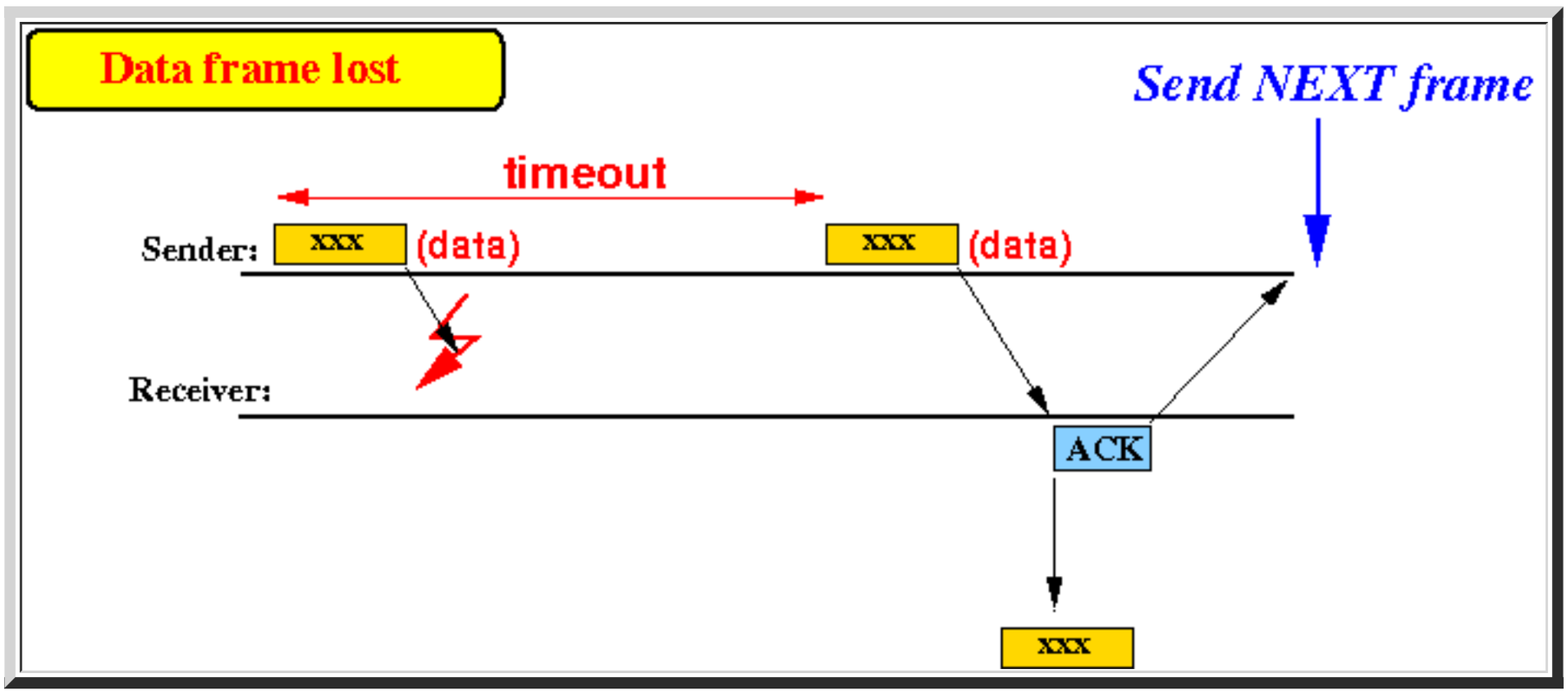
---

3. **Sender** receives the **ACK frame** before **timeout *expires***

   - **Done !!!**

- **Error scenario 1: data frame is *lost***

  - **Scenario 1** : **transmission error** in the ***data*** **frame**



According to the ***basic*** **protocol**:

1. The **sender** transmits **data frame**

   - **Sender sets** a ***timeout*** for **frame**

---

2. **Data frame** is ***lost*** (bit error(s))

   - **Receiver** will ***not*** transmit an **ACK frame** !!!

---

3. The **sender** will **time out**:

   - **Sender** will ***retransmit*** the ***same*** **frame**

---

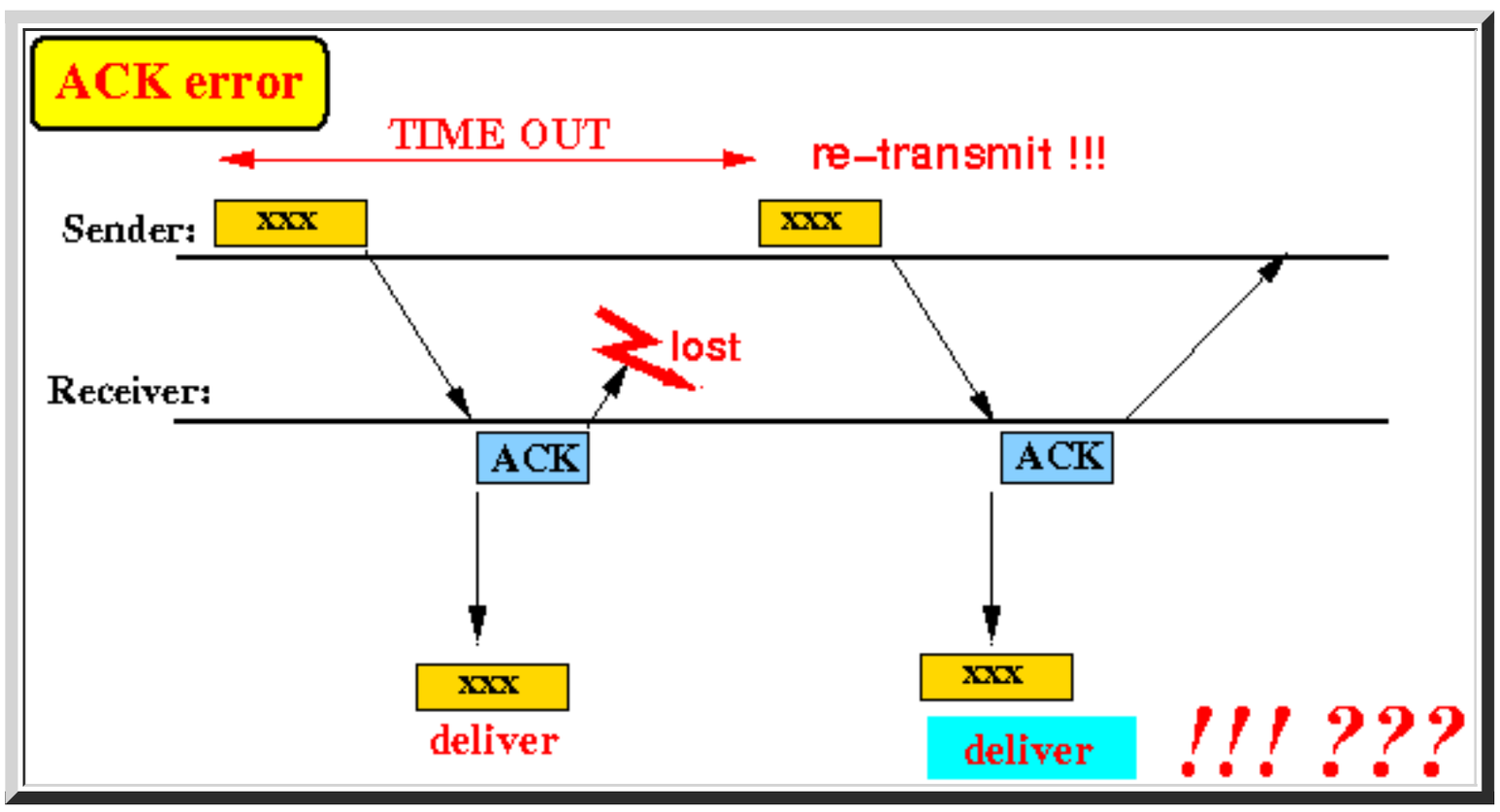
4. The time, the **receiver** recived it ***correctly*** and transmits a **ACK frame**

  - **Conclusion:**

    - The **simple protocol** can ***recover***:

      - **Data frame** transmission **errors** !!!

- **Error scenario 2: ACK frame is *lost***
  - **Scenario 2** : **transmission error** in the ***acknowledgement*** **frame**



According to the ***basic*** **protocol**:

1. The **sender** transmits the **data frame**

2. The **receiver** receives the **data frame** (correctly)

   - The **receiver** transmits an **ACK frame**
   - The **receiver** will ***deliver*** the **data frame**

3. The **ACK frame** is ***lost***.....

   - The **sender** will ***time out*** !!!

   Consequently:

   - The **sender** will ***assume*** that the **data frame** was ***lost*** !!!

4. The **sender** will ***re-transmit*** the ***same*** **data frame**

   - The **receiver** receives a **(*duplicate*) frame**
   - The **receiver** will **deliver** the **data frame**

   Error:

   - A **data frame** is **delivered *multiple* times** !!!!

Recall that:

- **Reliable communication** means:
  - **All messages** received **exactly *once***
  - (And in the **same order** as they were **transmitted**)

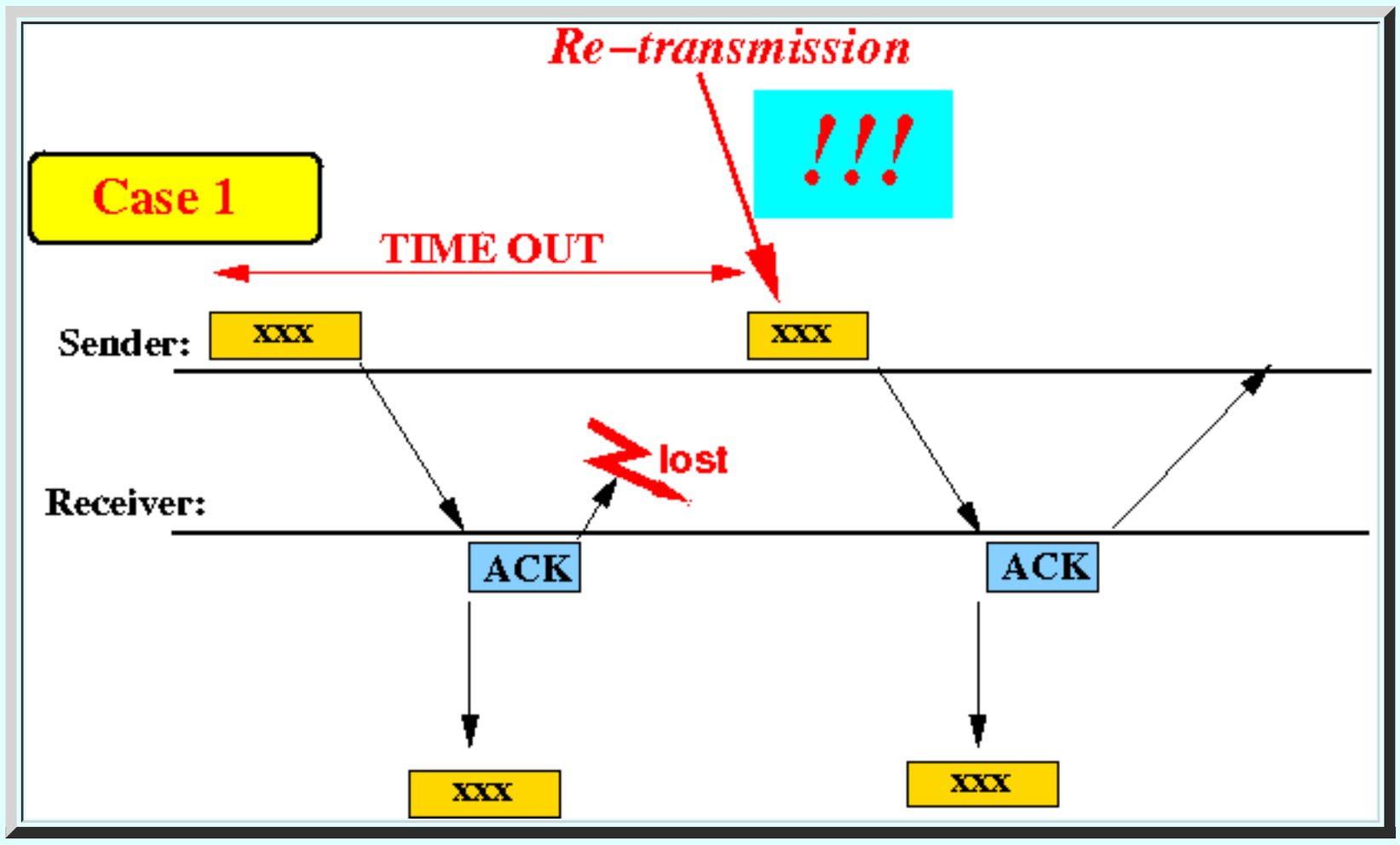
- **Fixing the protocol error**
  - **Right now**, we **cannot correct** the **protocol error because**:

    - The **receiver** is ***unable*** to ***distinguish***:

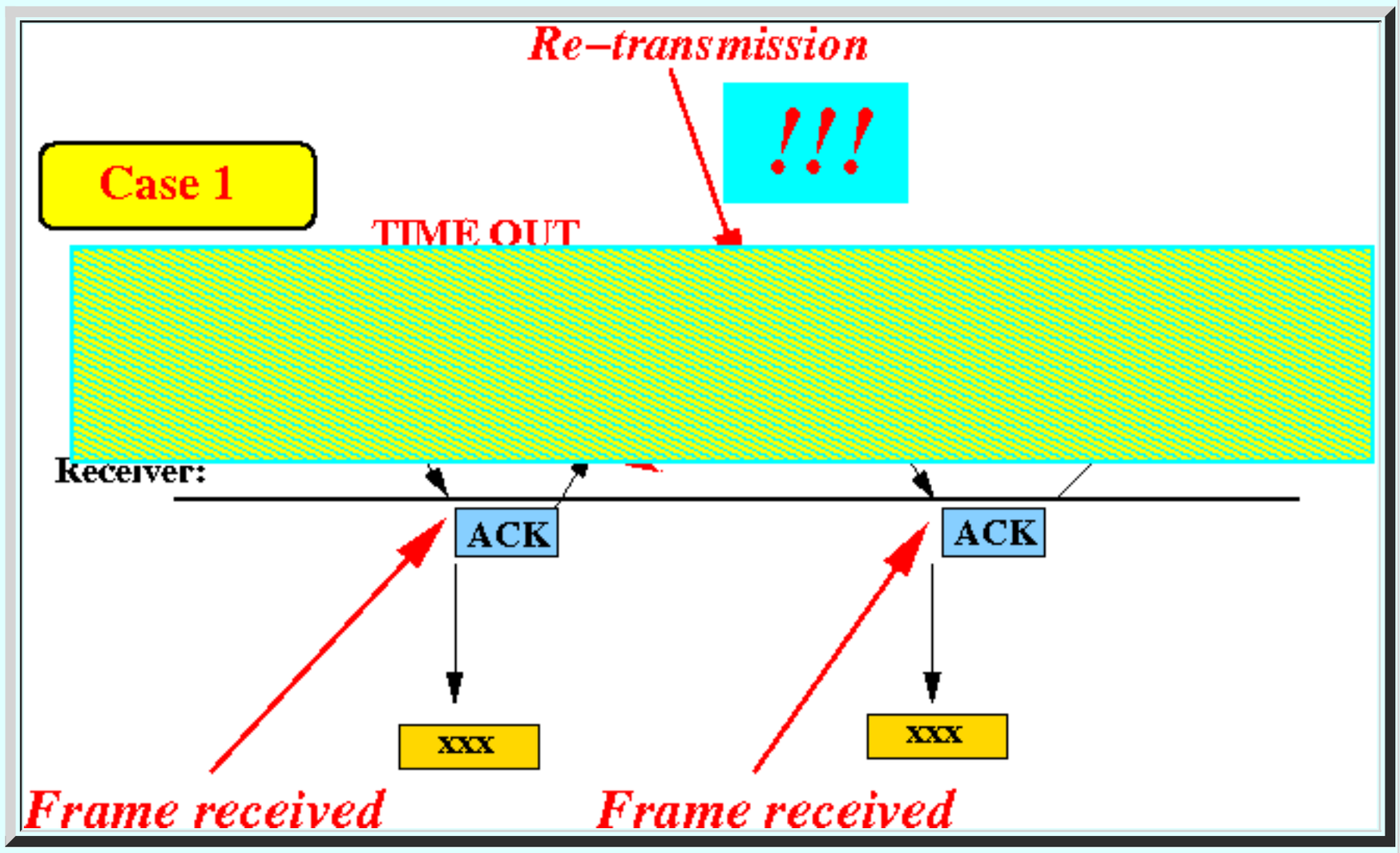
      - A ***re-transmission*** **(= old)** of a **data frame** **and**
      - A ***new*** **transmission** of a **data frame**

    - **Graphically explained:**

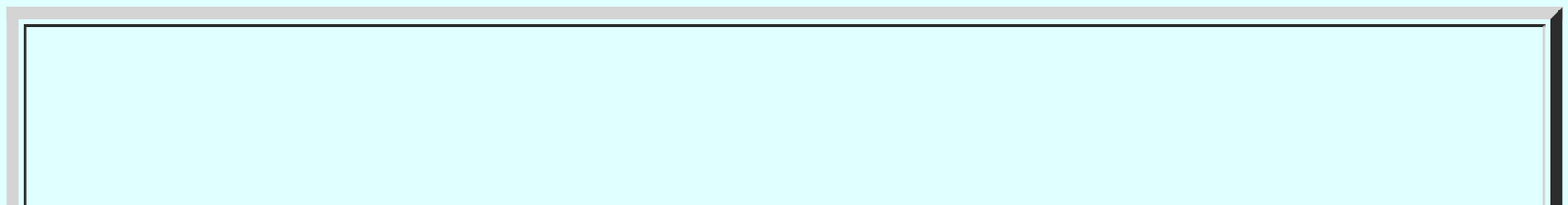
      - **Case 1:** ***re-transmission*** of an ***old*** **frame**

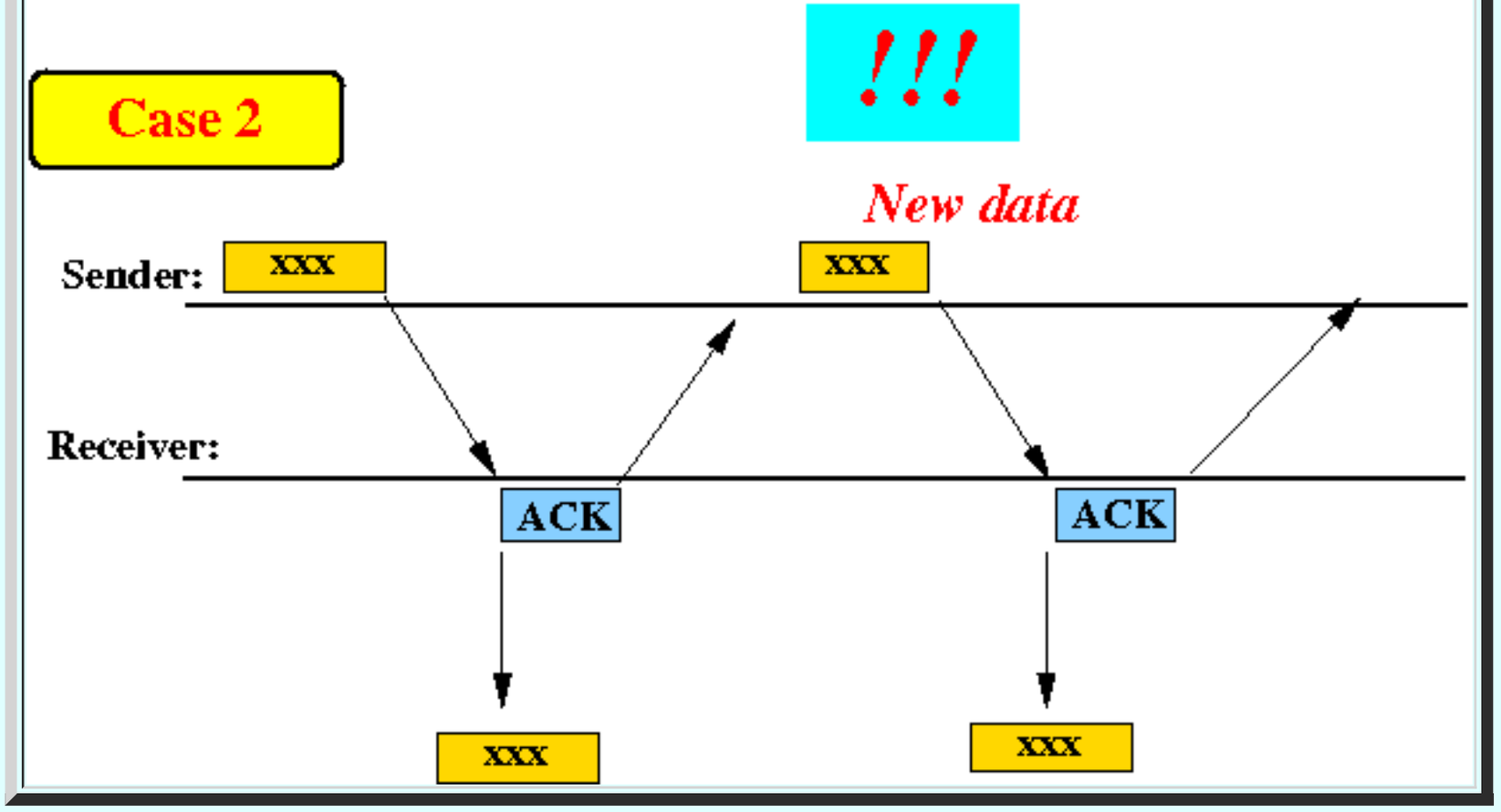


Remember, the ***receiver*** will ***only*** "see" the **following events**:

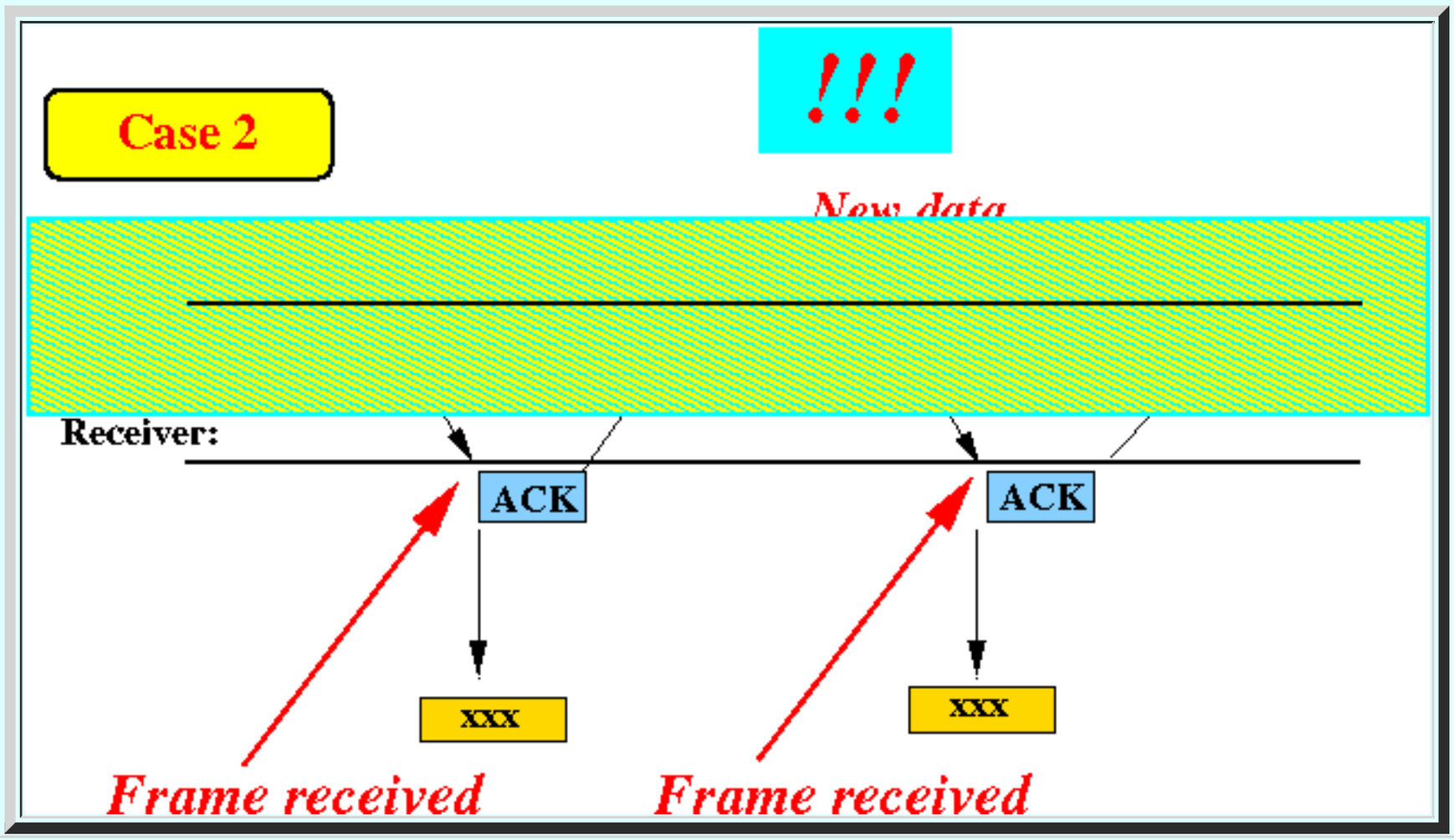


- **Case 2:**





Remember, the ***receiver*** will ***only*** "see" the **following events**:



The **receiver** do **not** have **enough information** to **distinguish** these ***2 different* cases** from **one another** !!!

- The **fix** (**add** more ***information*** !!!):

  - The **sender adds** a ***send* sequence number** to **identify** each **data frame *uniquely***

  - The ***receiver*** will **use** the ***send* sequence number** to **check** whether a **data frame** is:

    - a ***new* transmission** ***or***
    - a ***re-transmission***

Fix: adding sequence numbers to identify *data* frames

- ***Fixing* the basic protocol**
  - **Recall**: the **cause** of the **error**

    - The **receiver** ***cannot* tell** whether a **frame** is:

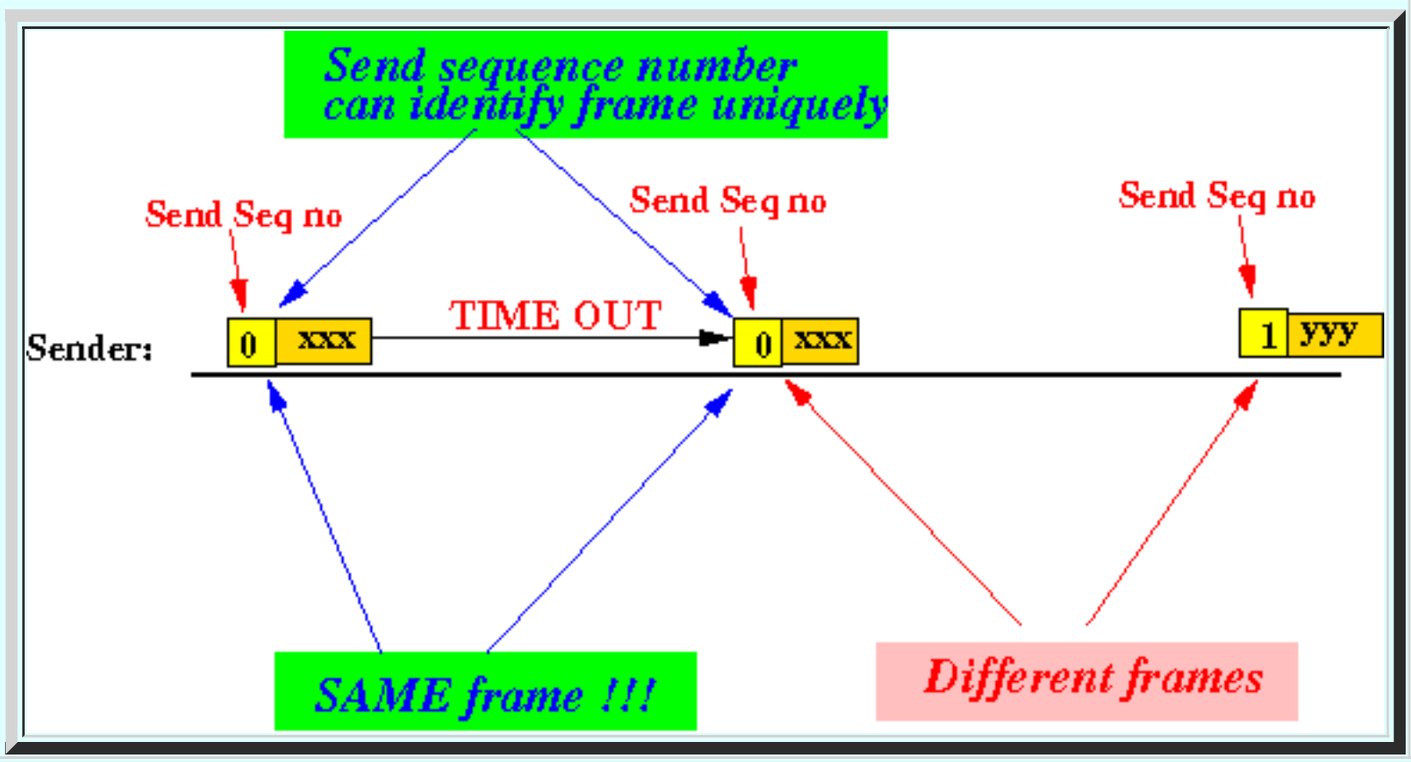
      1. A ***re-transmission*** **or**
      2. A ***now* transmission**

  - **Proposed *fix***

    - **Make** each **data frame *distinguishable*** by:

      - Adding a **unique *send* sequence number** to each **data frame**

    Example:



- **Simplifying assumption....**
  - **Assumption:**

    - **For now**, let us **assume**:

      - We can **use** an ***infinite* number** of ***different* sequence numbers**

    Note:

    - This **assumption** will make the **protocol *easy*** to **understand** and **prove**

    - We will **fix (drop)** this **assumption *later***

- **The *modified* basic reliable communication protocol**
  - **Assumption:**

    - **Sender** and **receiver** are **initialized** with the ***same*** `Frame#`

- Consider the following ***modified*** **protocol** to achieve **reliable transmission**:

  - **Sender:**

```
Start:
      Transmit frame with send seq no = Frame#;

      Set a timeout;

      while ( not timeout && ACK not received )
      {
         Wait for ACK frame;
      }

      if ( timeout expired )
      {
         go to Start;
      }

      Frame# = Frame# + 1;

      Done; (ACK was received !)
```

  - **Receiver:**

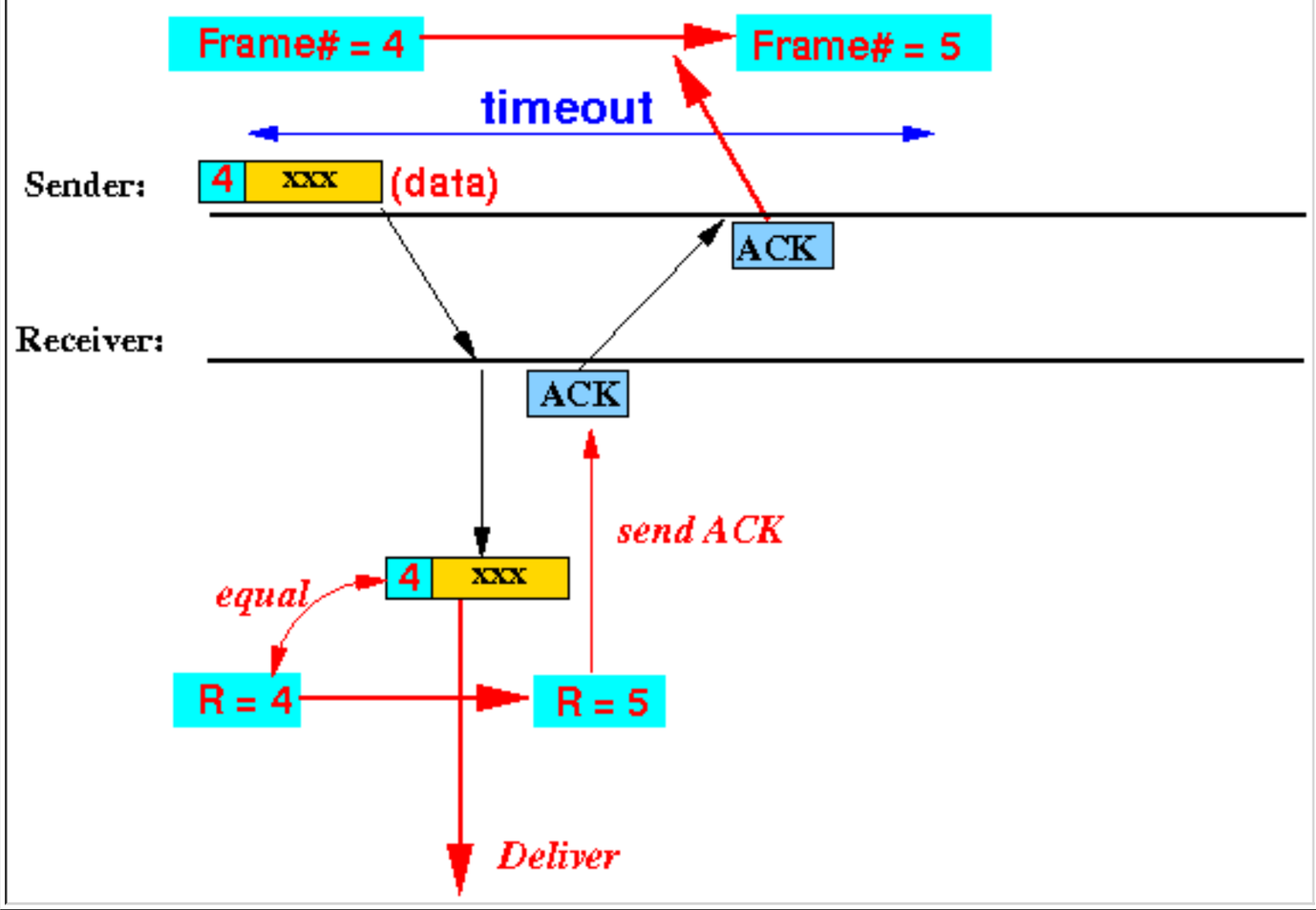
```
      // Receiver is ready to receive frame number R

      Receive frame;

      if ( CRC check is OK )
      {  /* ---------------------------------------------
            Receiver must only deliver new data frames
            --------------------------------------------- */
         if ( send seq no == R )
         {
            Send ACK frame to sender;   // ACK the new frame

            Deliver frame;

            R = R + 1;      // Accept next new frame
         }
         else
         {
            // An OLD frame was received !!!

            Send ACK frame to sender;  // Correct lost ACK frame error !!!

            Discard frame;    // because it's an old frame
         }
      }
      else
      {
         // Bit errors !!!

         Discard frame;
      }
```

- **The *modified* protocol in error-free operation**

  - **Example**: ***error-free*** operation

Modified Reliable Communication Protocol

Error Free Operation:



According to the ***modified*** **protocol**:

1. **Sender** transmits **data frame** with `Send seq no = 4`

   - Sender **sets** a **timeout** for **frame**

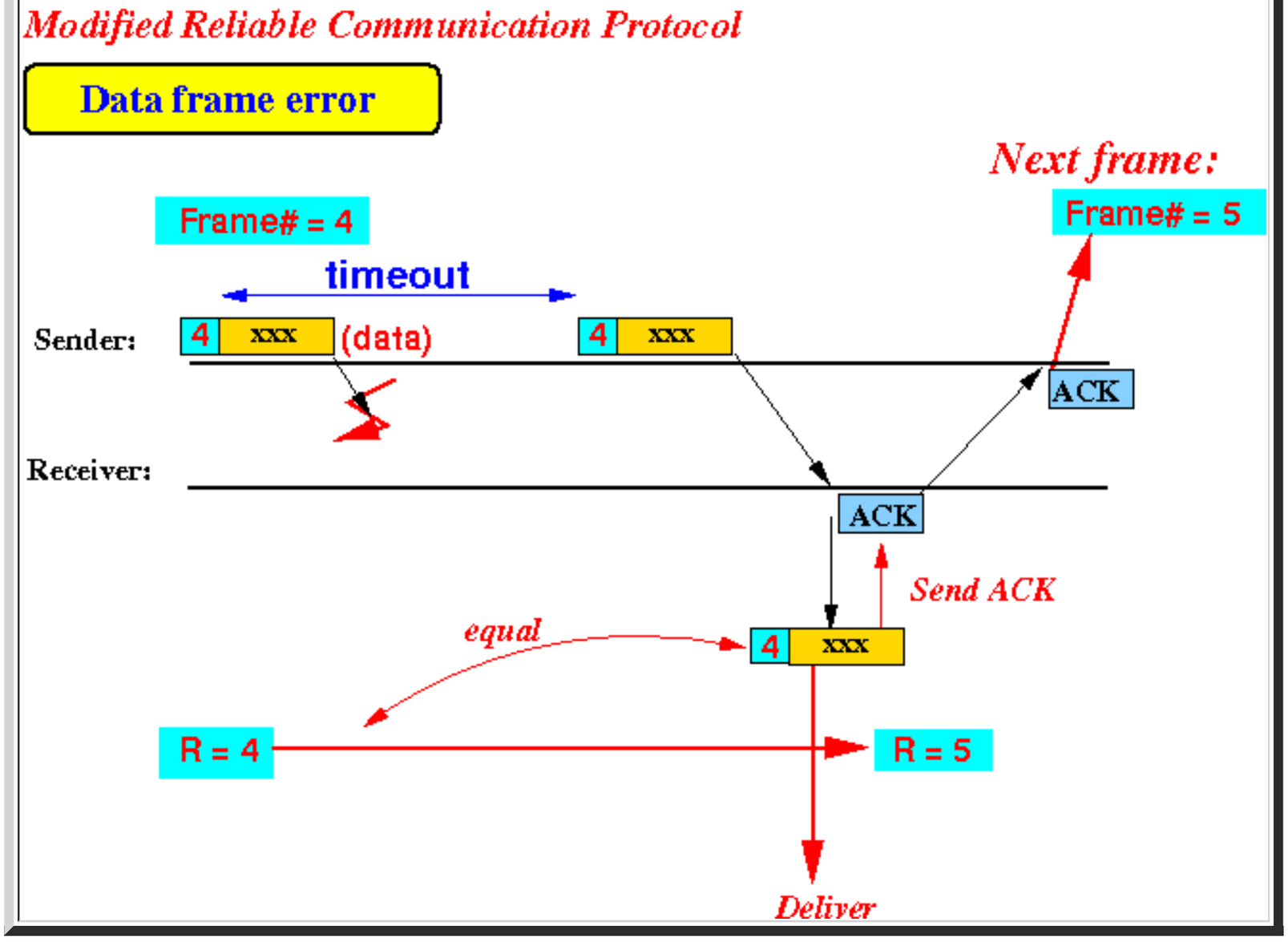
2. **Receiver** receives the **data frame** with ***matching*** `Frame#`:

   - **Receiver** transmits a **ACK frame**
   - **Receiver** ***increments*** `R = 5` !!!
   - **Receiver** delivers the **data frame**

3. **Sender** receives the **ACK frame** before **timeout** ***expires***

   - **Sender** ***increments*** `Frame# = 5` !!!
   - **Done** !!!

- **Error scenario 1: data frame is *lost***
  - **Scenario 1** : **transmission error** in the ***data*** **frame**



According to the ***modified*** **protocol**:

1. The **sender** transmits **data frame** with `send seq no = 4`

   - **Sender sets** a ***timeout*** for **frame**

2. **Data frame** is ***lost*** (bit error(s))

   - **Receiver** will ***not*** transmit an **ACK frame** !!!

3. The **sender** will **time out**:

   - **Sender** will ***retransmit*** the ***same*** **frame**

4. The **receiver** receives **Frame** and `send seq no == Frame#`:

   - **Receiver** transmits a **ACK frame**
   - **Receiver** ***increments*** `R = 5` !!!
   - **Receiver** delivers the **data frame**

5. **Sender** receives the **ACK frame** before **timeout** ***expires***

   - **Sender** ***increments*** `Frame# = 5` !!!
   - **Done** !!!

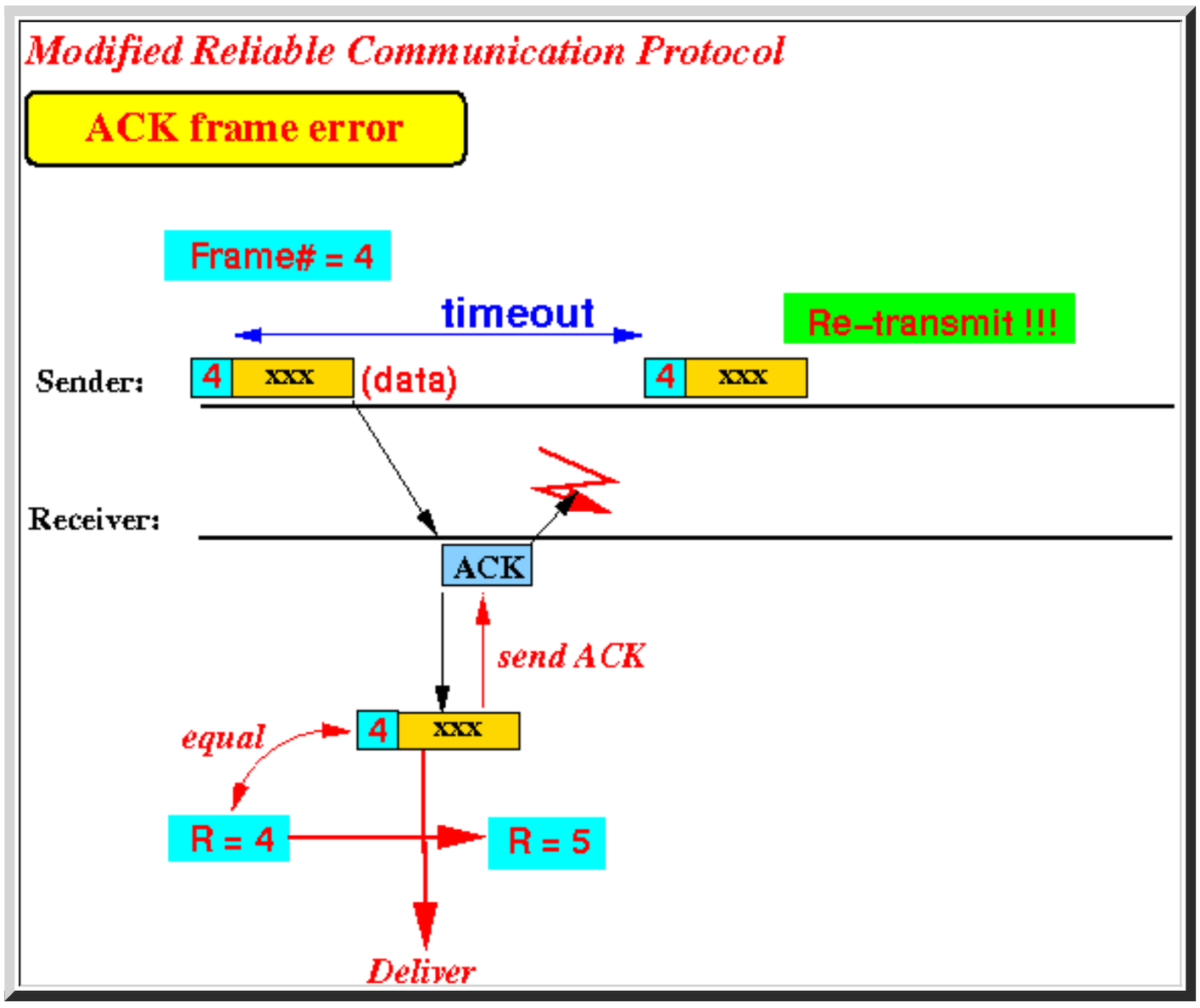
- **Conclusion:**

  - The ***modified*** **protocol** can ***recover***:

    - **Data frame** transmission **errors** !!!

- **Error scenario 2: ACK frame is *lost***

- **Scenario 2** : **transmission error** in the ***acknowledgement*** **frame**



According to the ***modified*** **protocol**:

1. **Sender** transmits **data frame** with `Send seq no = 4`

   - Sender **sets** a **timeout** for **frame**

---

2. **Receiver** receives the **data frame** with ***matching*** `Frame#`:

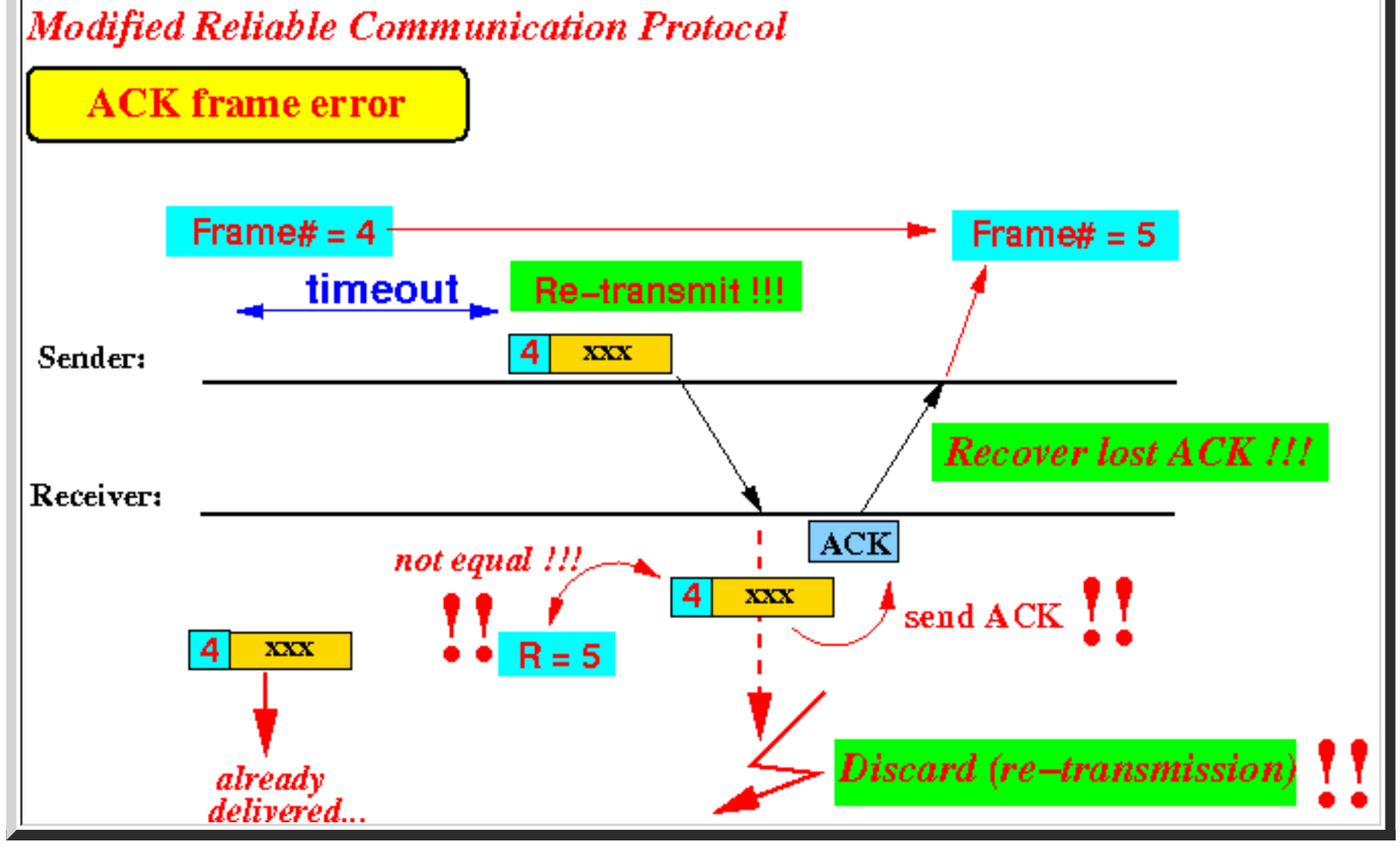
   - **Receiver** transmits a **ACK frame**
   - **Receiver** ***increments*** `R = 5` !!!
   - **Receiver** delivers the **data frame**

---

3. **ACK frame** is ***lost*** and the **sender** will **timeout**

   - **Sender** ***re-transmits*** `Frame# = 4` !!!

- The ***re-transmission*** is **processed** as **follows**:



According to the ***modified*** **protocol**:

1. **Sender** transmits **data frame** with `Send seq no = 4`

   - Sender **sets** a **timeout** for **frame**

---

2. **Receiver** receives the **data frame** with ***mismatched*** `Frame#` :

   - **Receiver** transmits a **ACK frame** (recovers ***lost*** **ACK** !!!)
   - **Receiver** ***discards*** the **data frame** (**duplicate** !!!)

---

3. **Sender** receives the **ACK frame** before **timeout** ***expires***

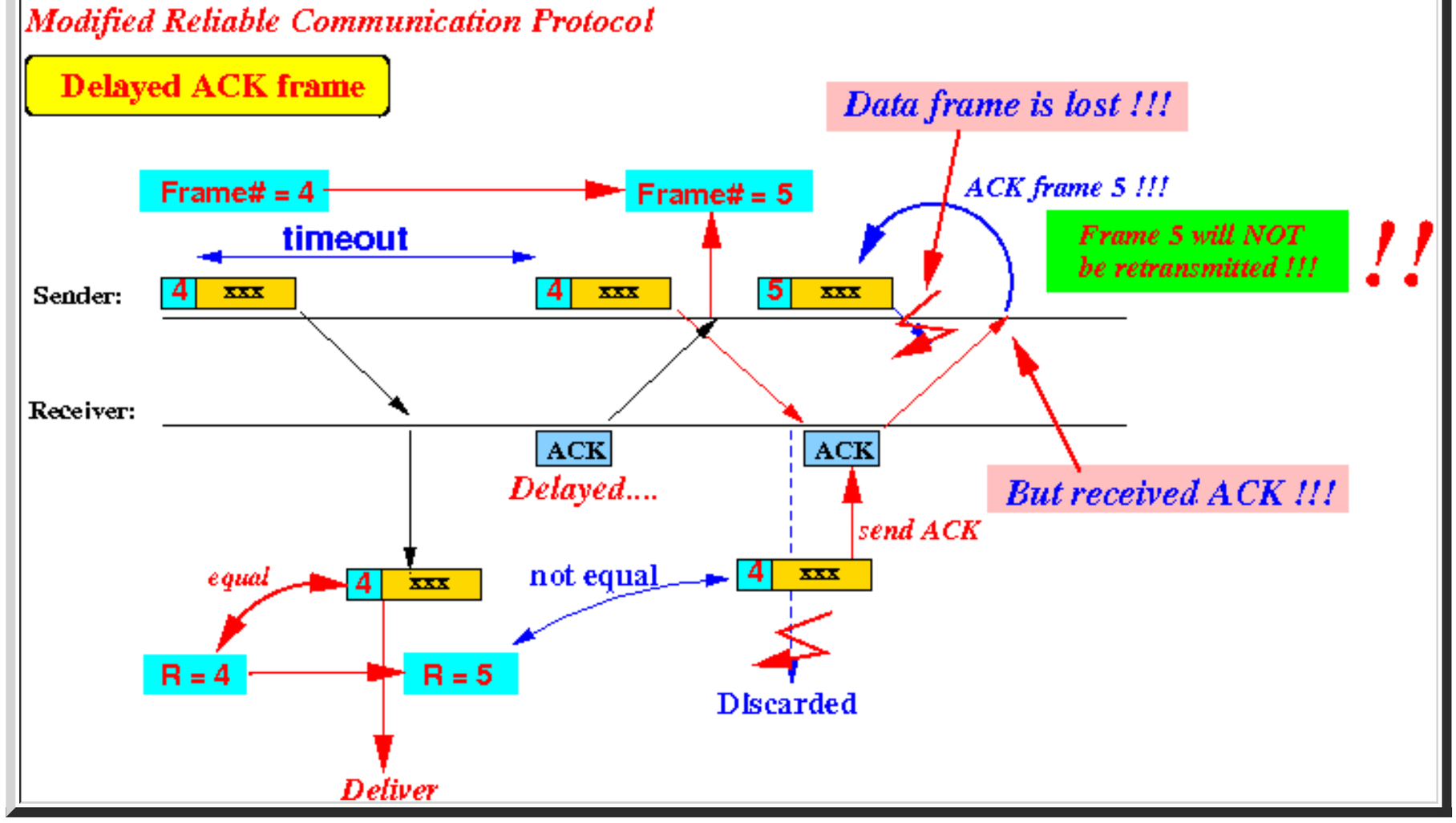
   - **Sender** ***increments*** `Frame# = 5` !!!
   - **Done** !!!

Correct !!!

- **Error scenario 3:** ***delayed*** **(or** ***duplicate*****) ACK frame**
  - **Reason** why an **ACK frame** can be **late**:

    1. The **receiving device** is **slow**
    2. The **receiver** was ***temproally*** **overloaded** (busy with other tasks)

  - **Error scenario 3**: **how** an ***last*** **ACK frame** can **cause** **error** in **reliable communication**:



Explanation:

1. **Sender** transmits **data frame** with `Send seq no = 4`

   - Sender **sets** a **timeout** for **frame**

---

2. **Receiver** receives the **data frame** with ***matching*** `Frame#`:

   - **Receiver** transmits a **ACK frame**
   - **Receiver *increments*** `R = 5` !!!
   - **Receiver** delivers the **data frame**

   But the **ACK frame** is ***delayed***

---

3. The **sender** will **time out**:

   - **Sender** will ***retransmit*** the **frame #4**

---

4. The **sender** receives the ***delayed*** **ACK**:

   - **Sender *increments*** `Frame# = 5` !!!

---

5. The **sender** **now** transmits a **new frame** with `send seq no = 5`

   - Sender **sets** a **timeout** for **frame**

   This ***new*** **data frame** is ***lost*** !!!

---

6. In the mean time, the ***re-transmission*** of **frame #4** is **received** at the ***receiver***

   The **receiver** finds `send seq no ≠ 5` and **detects (correctly)** that it is an ***old*** **frame**:

   - **Receiver** transmits a **ACK frame**
   - **Receiver *discards*** the **data frame** (**duplicate** !!!)

---

7. The **sender** receives the **ACK frame**:

   - **Sender *increments*** `Frame# = 6` !!!

(**Sender** thinks that **data frame #5** has been ***received*** !!!)

Error !!!

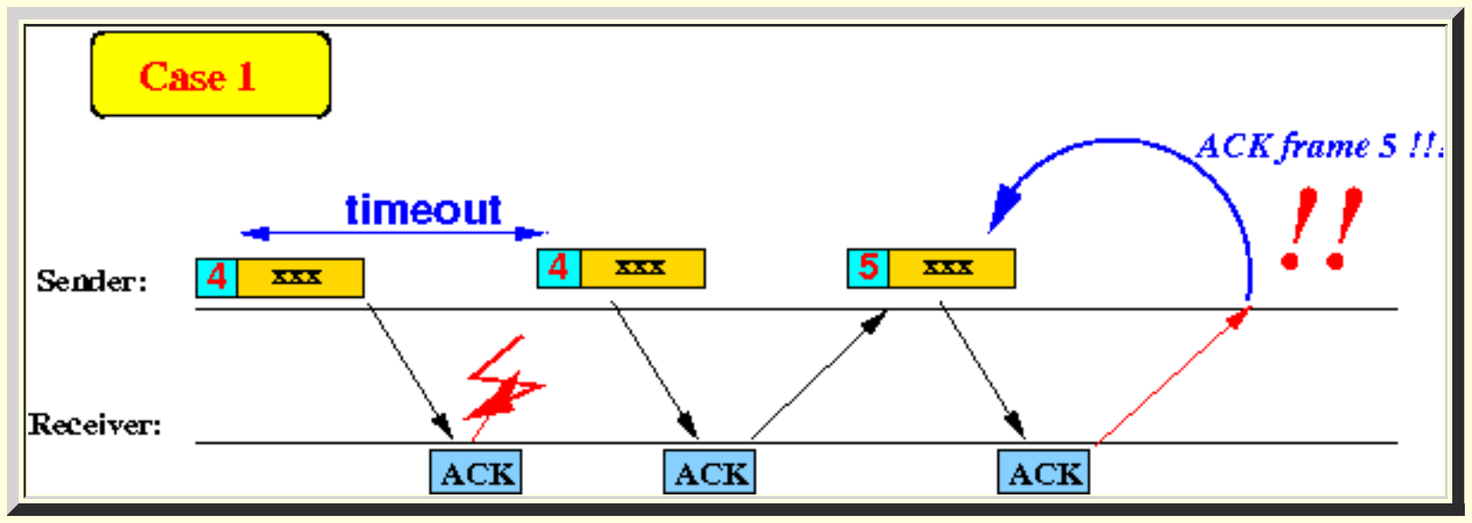
- **Fixing the protocol error**
  - **Right now**, we **cannot** fix the **protocol error because**:

    - The ***sender*** **cannot distinguish** between these **2** ***different*** **cases**:

      - An **acknowledgement** for an ***new*** **data frame**
      - An **acknowledgement** for an ***old*** **data frame**

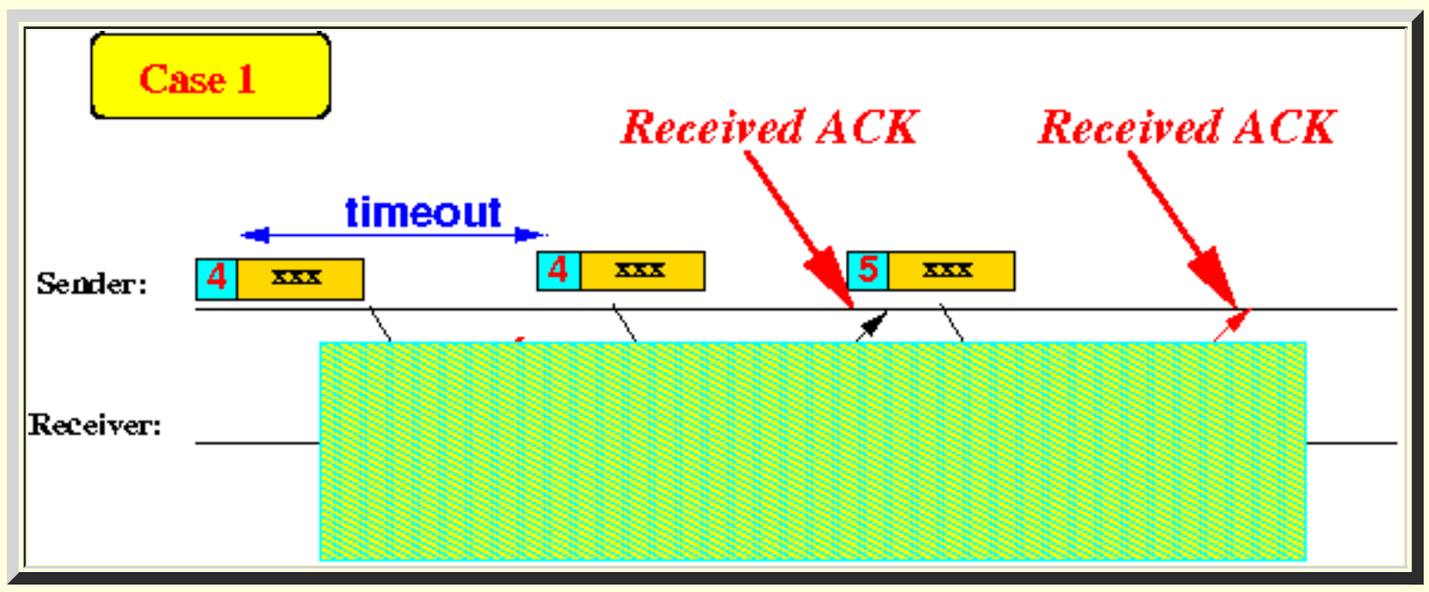
    - **Graphically explained:**

      - **Case 1:** the ***first*** **ACK** for **frame 4** was **lost**:

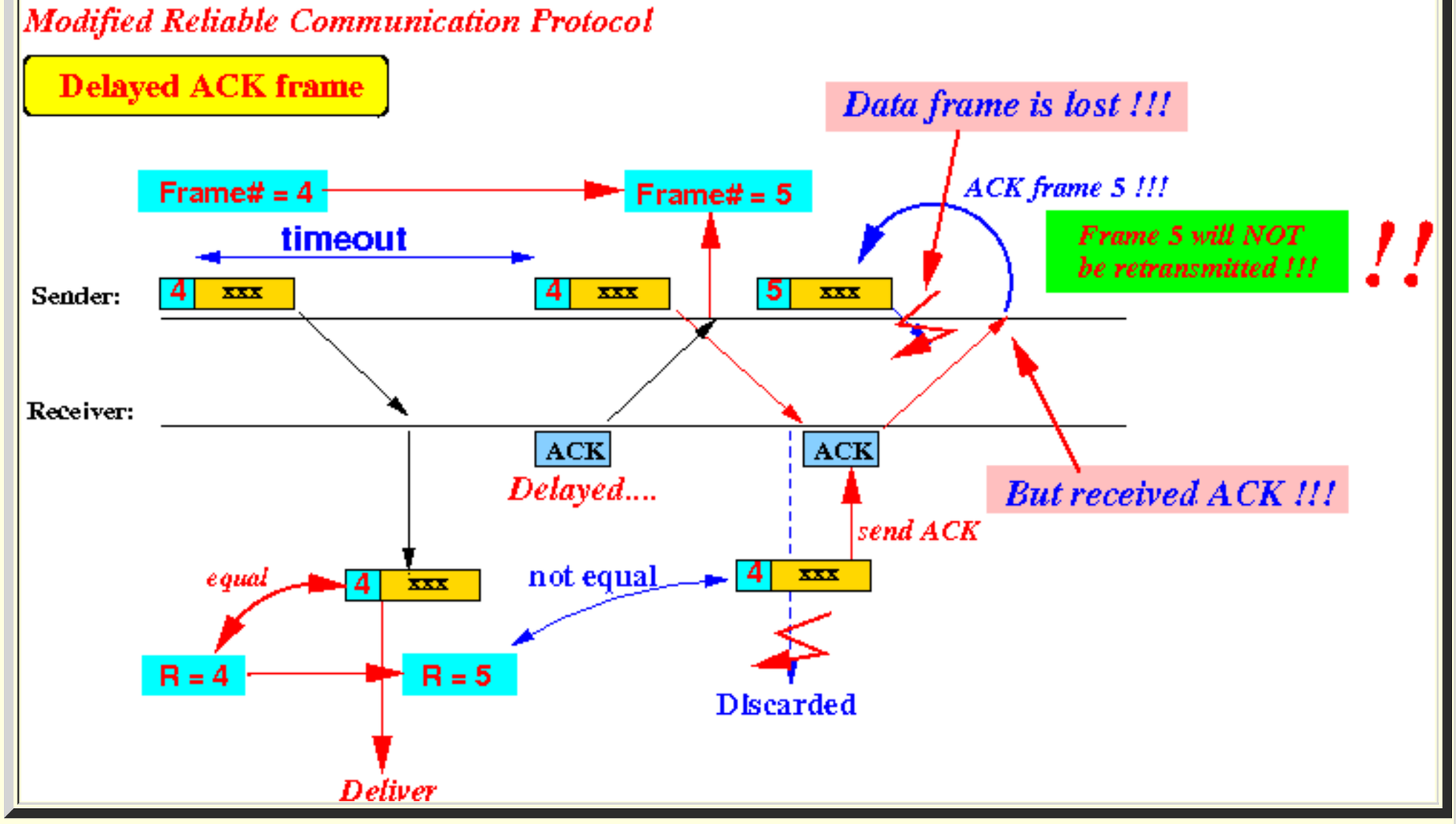


The **2nd** ***received*** **ACK acknowledges** the ***new*** **frame (#5)** ***correctly*** !!!

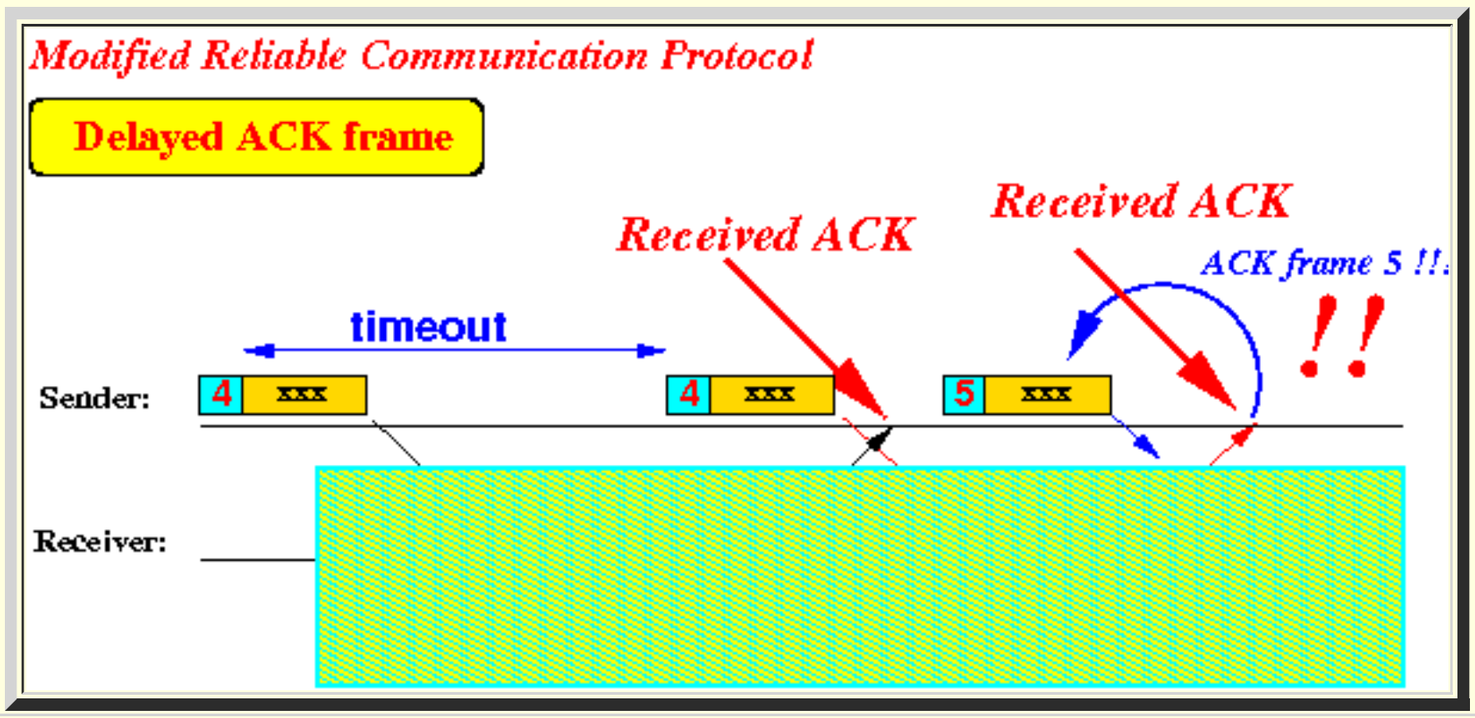
Remember that the ***sender*** will ***only*** "see" these **events**:



- **Case 2**: a ***delayed ACK*** **frame** was **used** to **acknowledge** the ***wrong*** **data frame**:



Remember that the ***sender*** will ***only*** "see" these **events**:



- **Fixing** the ***delay/duplicate*** **ACK problem:**

  - The **receiver adds** a ***recv*** **sequence number** to **identify** each **ACK frame** ***uniquely***

    - The ***recv*** **sequence number** in an **ACK frame** must **identify** the ***data*** **frame** that it is **"*acking*"**

  - The ***sender*** will **use** the ***recv*** **sequence number** in the **ACK frame** to **check** whether a **data frame** is ***acked***

Final fix: adding sequence numbers to *ACK* frames

- **The *Stop-and-Wait* protocol**
  - **Assumption:**
    - **Sender** and **receiver** are **initialized** with the ***same*** `Frame#`
  - The **Stop-and-Wait** protocol:
    - **Sender:**

```
Start:
      Transmit frame with send seq no = Frame#;

      Set a timeout;

Wait:
      // Wait for an ACK frame to arrive before time out...
      while ( not timeout && ACK not received )
      {
         Wait for ACK frame;
      }

      if ( timeout expired )
      {
         go to Start;
      }

      // ACK was received... Check for correct ACK
      if ( ACK seq no ≠ Frame# )
      {
         go to Wait;       // Wrong ACK !
      }

      Frame# = Frame# + 1;
      Done;
```

    - **Receiver:**

```
    // Receiver is ready to receive frame number R

    Receive frame;

    if ( CRC check is OK )
    {
       if ( send seq no == R )
       {
          Send ACK frame with recv seq no = R to sender;

          Deliver frame;

          R = R + 1;      // Accept next new frame
       }
       else
       {
          // Received an OLD data frame

          Send ACK frame with recv seq no = (R-1) to sender;

          Discard frame;    // because it's an old frame
       }
    }
```

```
else
{
   // Bit errors !!!

   Discard frame;
}
```

Special attention::

- When the **receiver** receives an ***old*** **frame**:
  - the **receiver** ***must*** send an **ACK frame** to **recover** the **ACK frame loss**

- To ***distinguish*** an **old ACK frame** and the ***current*** **ACK frame (= R)**, the **receiver** can use the value:

  (R-1)

- After **making** these ***2*** **modification** to the (my) ***basic*** **protocol**:

  - **Add** a ***Send*** **Sequence Number** to the **data frames (messages)** to:
    - **Identify** a ***data*** **frame** ***uniquely***

  - **Add** a ***Receive*** **Sequence Number** to the **ACK frames** to:
    - **Identify** a ***ACK*** **frame** ***uniquely***

  the **resulting protocol** is the **called**:

  - ***Stop-and-Wait*** protocol.

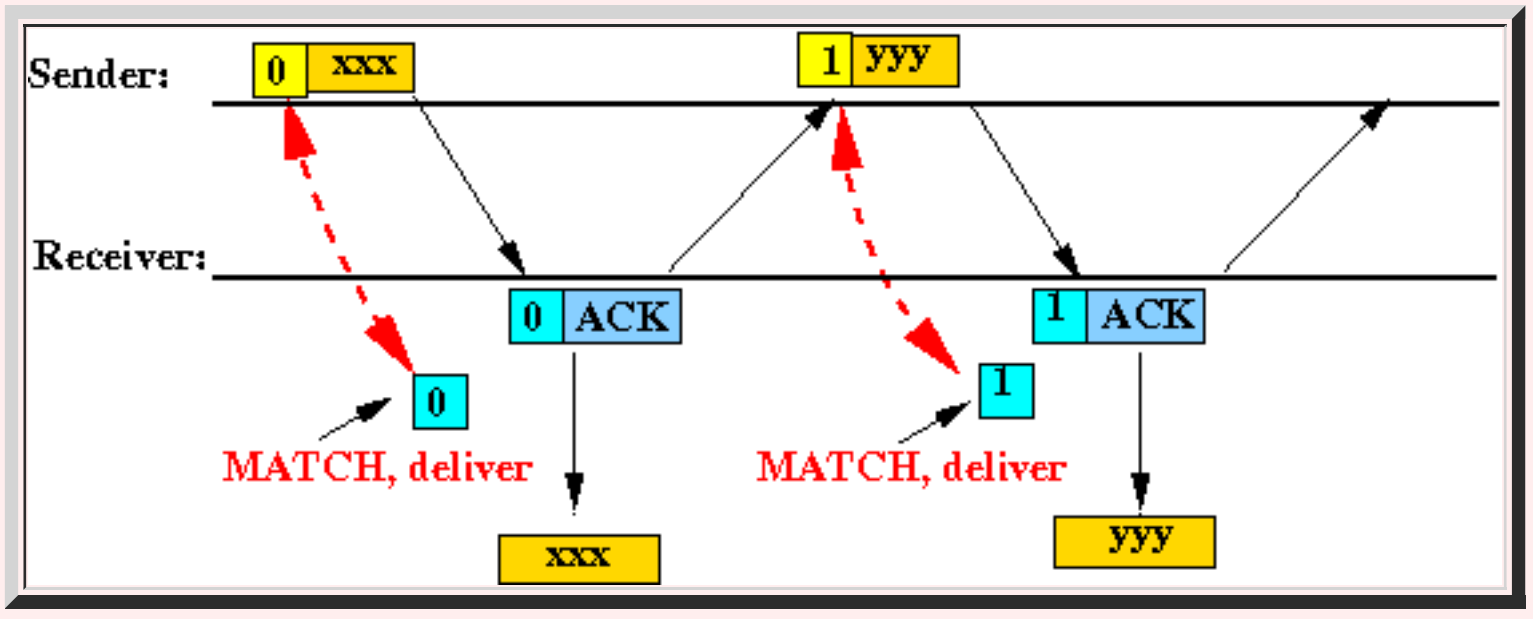
    (Stop-and-Wait is one of the "classic" reliable communication protocols).

- **The *Stop-and-Wait* protocol recovers from a delayed ACK**
  - **Note:**

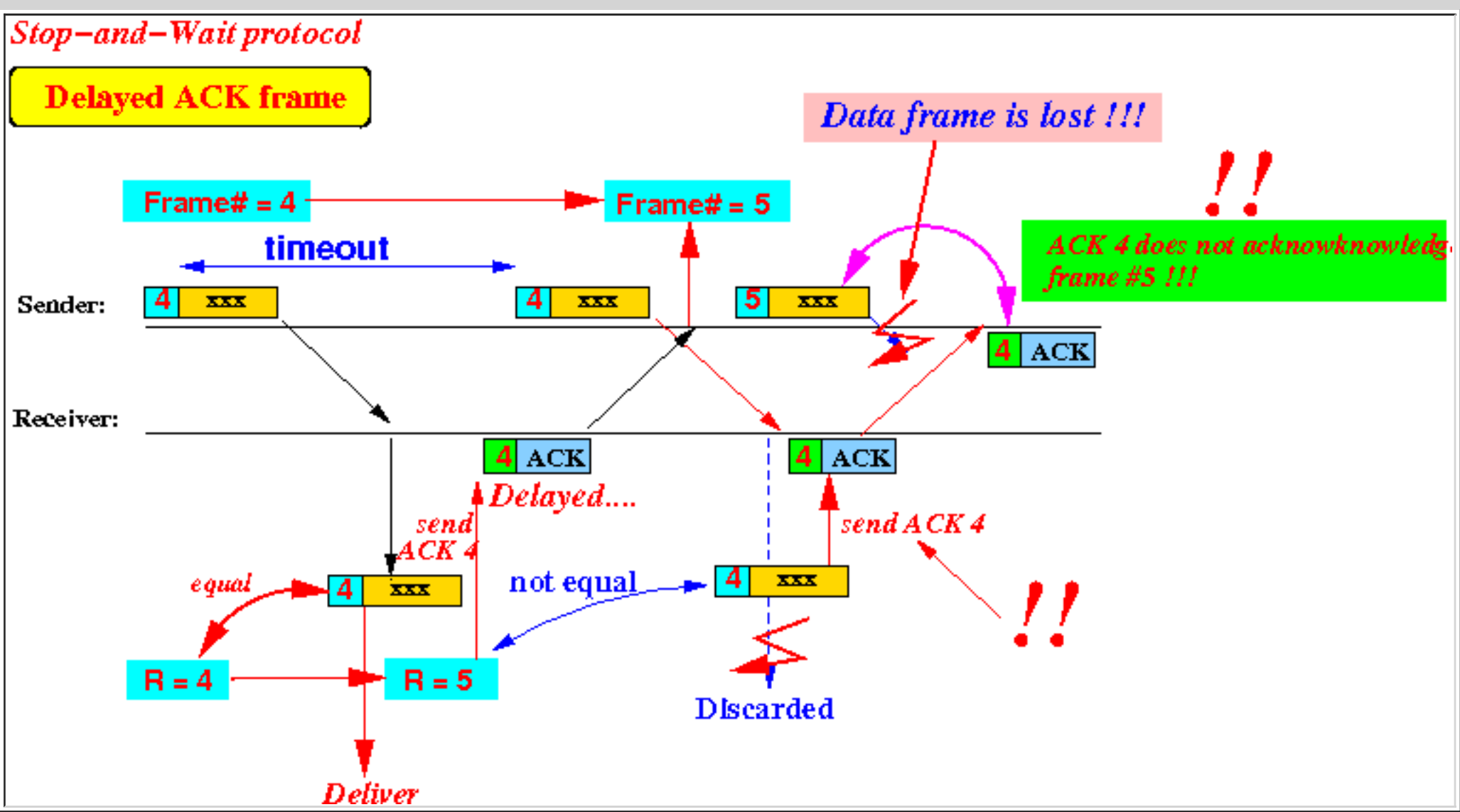
    - I will **skip** the following **scenarios**:
      - **Error-free**
      - **Data frame** is **lost**
      - **ACK frame** is **lost**

    because: the **operation** of the **Stop-and-Wait** protocol is ***similar*** to **that** of the ***modified*** **basic protocol**

- **Example:** when there are **no erros**, the the **sender** and **receiver** will operate like this:



- **How** the ***Stop-and-Wait*** protocol **solves** the ***delayed*** **ACK problem:**



Explanation:

- When the **receiver** receives the ***2nd*** **frame #4**, the **receiver** will **detect** that:

  - this **frame #4** is a ***re-transmission*** **because**:

    - ***Send*** **sequence no = 4**
    - **R = 5 !!!!**

- The **receiver** will **transmit**:

```
ACK 4     // (R-1) = 5-1 = 4
```

- When the **sender** receives the **ACK 4**, the **sender** finds:

  - **ACK no = 4**
  - **Frame# = 5**

  The **sender** can **tell** the the **ACK 4** is **not** **acknowledging** the **data frame 5**

  - The **sender** will **continue** waiting for **ACK 5**
  - In fact, the **sender** will ***timeout*** and ***retranmits*** the **lost frame #5** again (and recovers correctly)

- **Postscript: full name**

  - The **complete name** of the **Stop-and-Wait** protocol is:

    - **Stop-and-Wait Automatic Repeat reQuest** protocol

      Or: **Stop-and-Wait *ARQ*** protocol

      Wikipedia page: click here

- **ARQ protocols**

  - **ARQ protocols:**

    - **ARQ protocol** = a ***family*** of **error-control methods** for **data transmission**

    - **ARQ protocols** use:

      - **ACK frames** **and**
      - **timeouts**

      to **achieve *reliable* data transmission** over an ***unreliable* transmission medium**.

      Wikipedia page: click here

  - **Stop-and-Wait**:

    - The **Stop-and-Wait ARQ** protocol is the ***simplest* member** of the **ARQ protocols**

  - We will **soon** study a **more complex** (and more **efficient**) **ARQ** protocol:

    - The **Sliding Window ARQ** protocol

Performance of the communication protocol

- **Efficiency: Channel utilization**
  - **Channel utilization**
    - **Channel utilization** = the **fraction** of the **transmission capacity** of the **communication channel** that contains ***data*** **(frames) transmissions**

$$\text{Channel utilization} = \frac{\text{Time used to transmit data frames}}{\text{Total amount of time}}$$

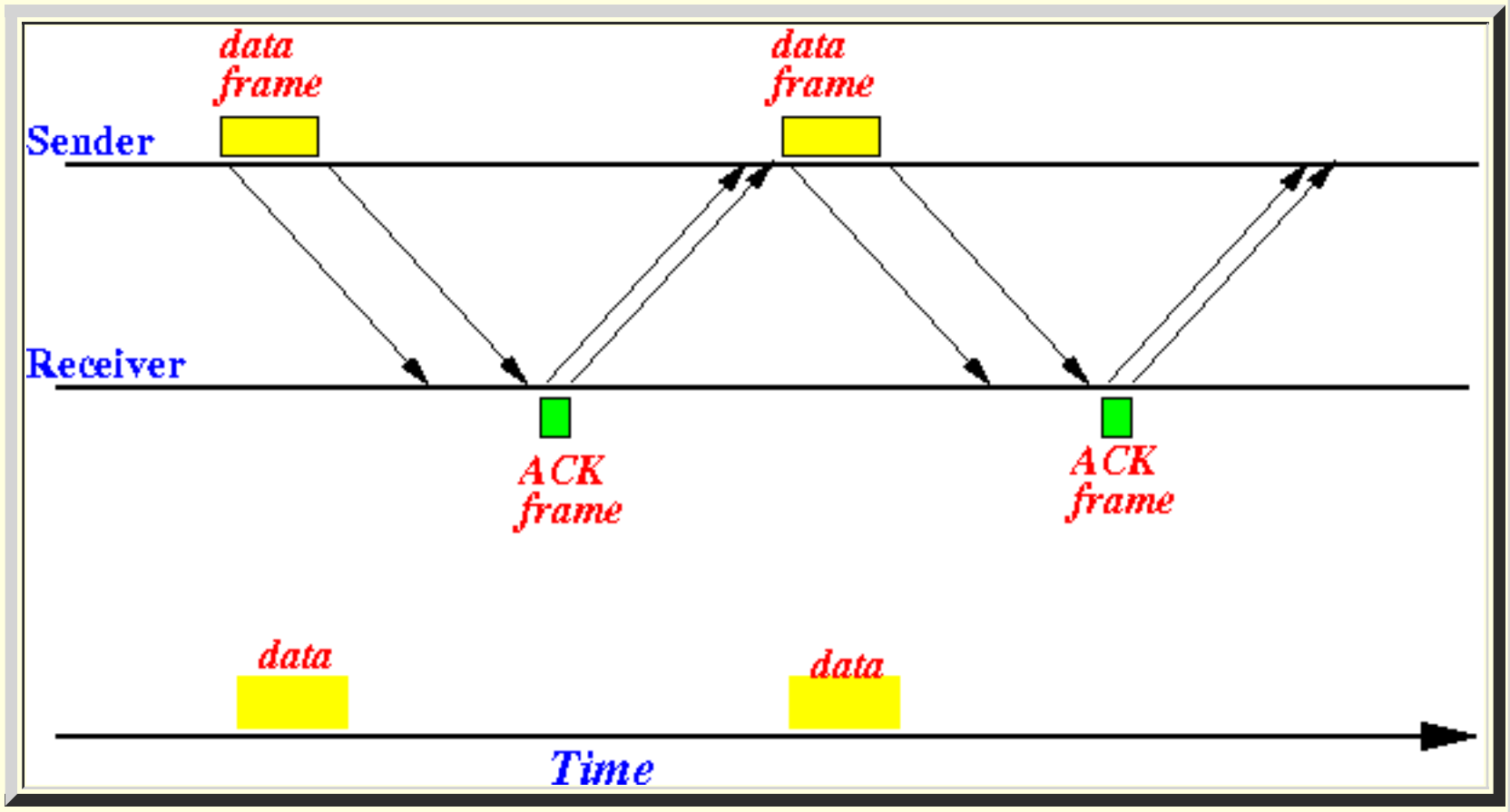
  - **Another term** for **channel utilization**:
    - **Channel *efficiency*** or simply **efficiency**

- **Important fact from Physics**
  - **Facts:**
    - **Speed** of **light** in **vacuum (or space)** ~= **3×10^8 m/sec**
    - **Speed** of **electrical signal** in **(copper) wire** ~= **2×10^8 m/sec**

Maximum channel utilization of the Stop-and-Wait protocol

- **Performance of Stop-and-Wait protocol**
  - **Question:**

    - A **sender** transmits **data frames** to a **receiver** using the **Stop-and-Wait** protocol:



    - What is the **maximum channel utilization** ?

      - In other words: what ***fraction*** of **time** on the **time line** in the **above figure** is ***yellow*** (= data frames)

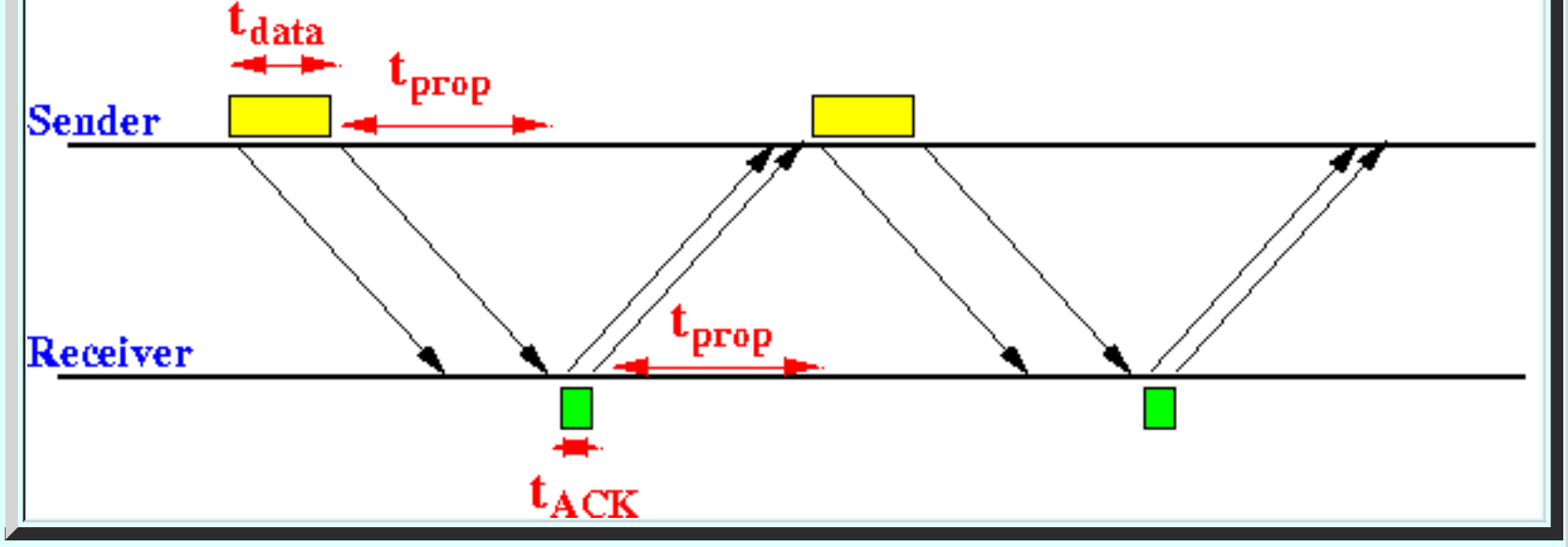
Note:

- The ***maximum*** **channel utilization** is **experienced** when there are ***no*** **transmission errors**

  - When there are **transmission errors**, there will be ***extra*** **overhead** to **recover** from the **error(s)**

  (So the **question implies** that we must **assume** that there are ***no*** **transmission errors**)

- **Notations used in the performance analysis**
  - **Notations:**



- t_{data} = time to **transmit** the ***data* frame**

- t_{prop} = time for the signal to **propagate** from the **sender** to the **receiver**

  - This **delay** is called the **propagation delay**

- t_{ACK} = time to **transmit** the **ACK frame**

- **NOTE:**

  - the **ACK frame** is **usually** **very short** compared to a **data frame**

  - **Therefore:**

    $$\frac{t_{ACK}}{t_{data}} << 1$$

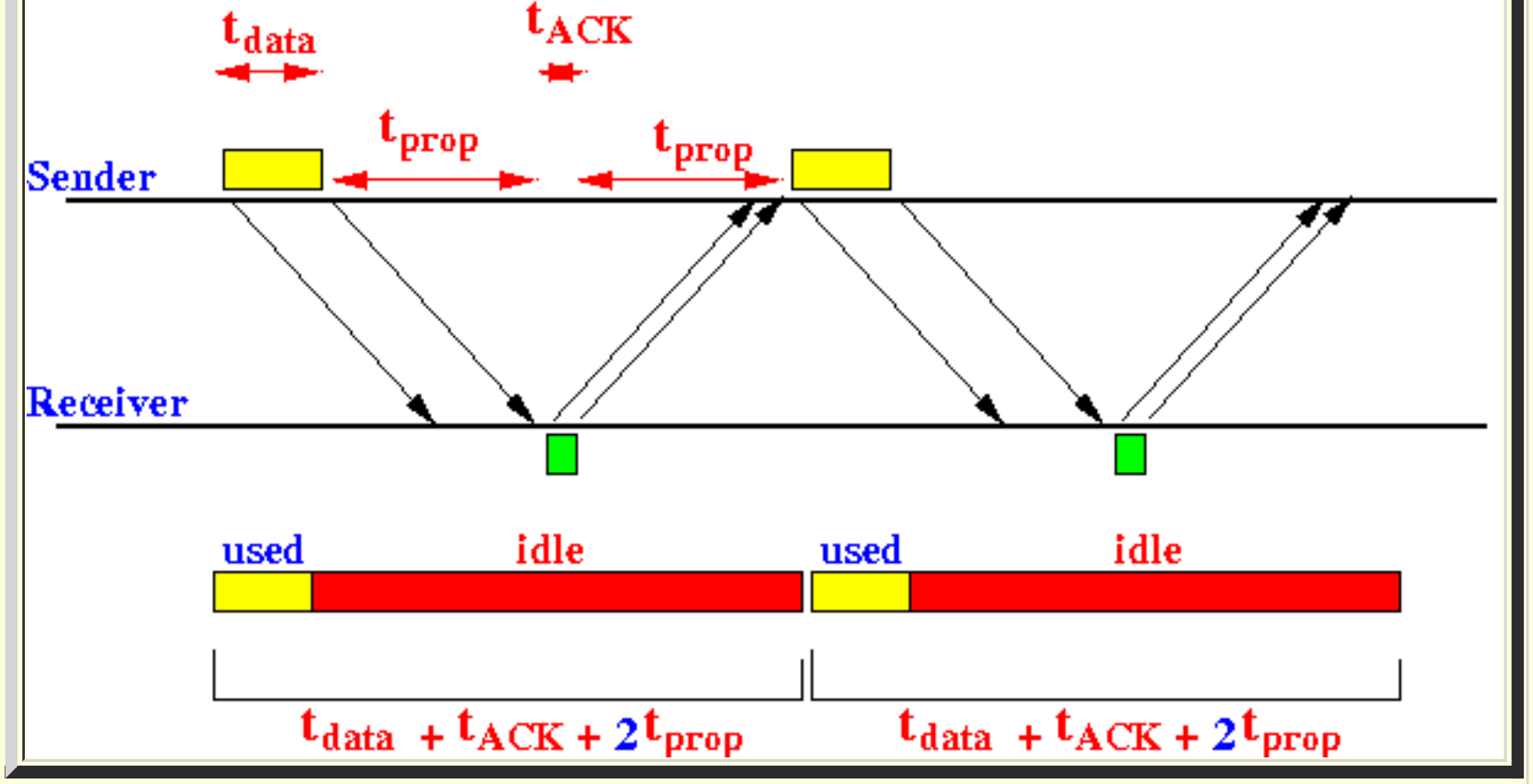
  - It is often **assumed** that:

    $$\frac{t_{ACK}}{t_{data}} = 0$$

- **Maximum channel utilization of the Stop-and-Wait protocol**

  - **Maximum channel utilization** of the **transmission channel**:

    - The following **pattern** is **repeated** **indefinitely** when there are ***no* transmission errors**:



- The ***duration*** of **each *cycle*** is:

$$t_{data} + t_{prop} + t_{ACK} + t_{prop}$$
$$= t_{data} + t_{ACK} + 2 \times t_{prop}$$

- The ***amount* of time** in the **cycle** used to transmit ***data* frames** is:

$$t_{data}$$

- The ***fraction*** of **time** used to **transmit *data* frames** is:

$$\text{Channel efficiency} = \frac{t_{data}}{t_{data} + t_{ACK} + 2\ t_{prop}}$$
$$\sim= \frac{t_{data}}{t_{data} + 2\ t_{prop}}$$

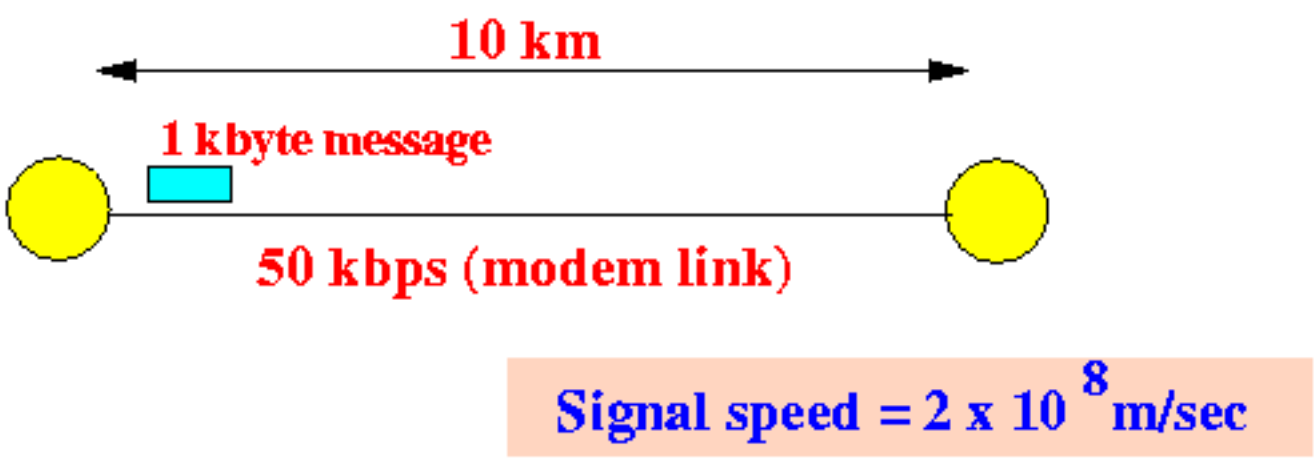
(t_{ACK} is ***very* small** and is usually **discarded** in the approximation)

Performance of the Stop-and-Wait protocol in low and high data rate links

- **History of the *Stop-and-Wait* protocol**
  - **Fact:**
    - The **Stop-and-Wait** protocol was ***very*** **popular** in the **1990s**
    - **Nowadays**, the **Stop-and-Wait** protocol is ***not*** **used at all** !!!
  - A bit of **history** first:
    - Back in **1995**, we used **modems** from **home** to **connect** workstation on **campus**
      - The **transmission rate** of **modems** were about **56 kbps**
    - **Nowadays**, you have **DSL** and **Cable** **Internet service** that provides:
      - **150 Mbps ~ 600 Mbps**

    After **computing** the ***maximum*** **channel utilization** of **Stop-and-Wait**, you will understand **why** the **Stop-and-Wait** protocol is ***no longer*** used **today** !

- **Performance of Stop-and-Wait on low data rate (bandwidth) links**
  - **Case study 1:** efficiency of ***Stop-and-Wait*** on a **low speed** **transmission link**
    - **Find** the ***maximum*** **channel utilization** of **Stop-and-Wait** in the following **transmission link**:



Summary of the **operational parameters**:

- **Data rate** of the **transmission link** = **50 kbps (kilobits/sec)**
- The **length** of the **link** = **10 km**

- **Sender** transmits **data frames** of size **1 kBytes**
- We will **ignore** the **time** needed to **transmit** an **ACK frame**

because: $t_{ACK} << t_{data}$

- **First**: compute t_{data}, t_{prop} and t_{ACK}.

- Compute t_{data}:

```
1 data frame  = 1 kbytes
              ~= 1000 bytes
               = 8000 bits

Transmission speed  = 50 kps
                    ~= 50000 bits in one sec

===> time to transmit 1 data frame = 8000/50000
                                   = 0.16 sec

     t_data = 0.16 sec
```

- Compute t_{ACK}:

  - We *assume* that $t_{ACK} \sim= 0$

- Compute t_{prop}:

```
Signal speed = 2 × 10^8 m/sec
             = 2 × 10^5 km/sec

Distance between sender and receiver = 10 km

===> time for signal to travel 10 km = 10 /(2 × 10^5)
                                     = 5 × 10^-5

     t_prop = 5 × 10^-5 sec
            = 0.00005 sec
```

- **Maximum channel utilization:**

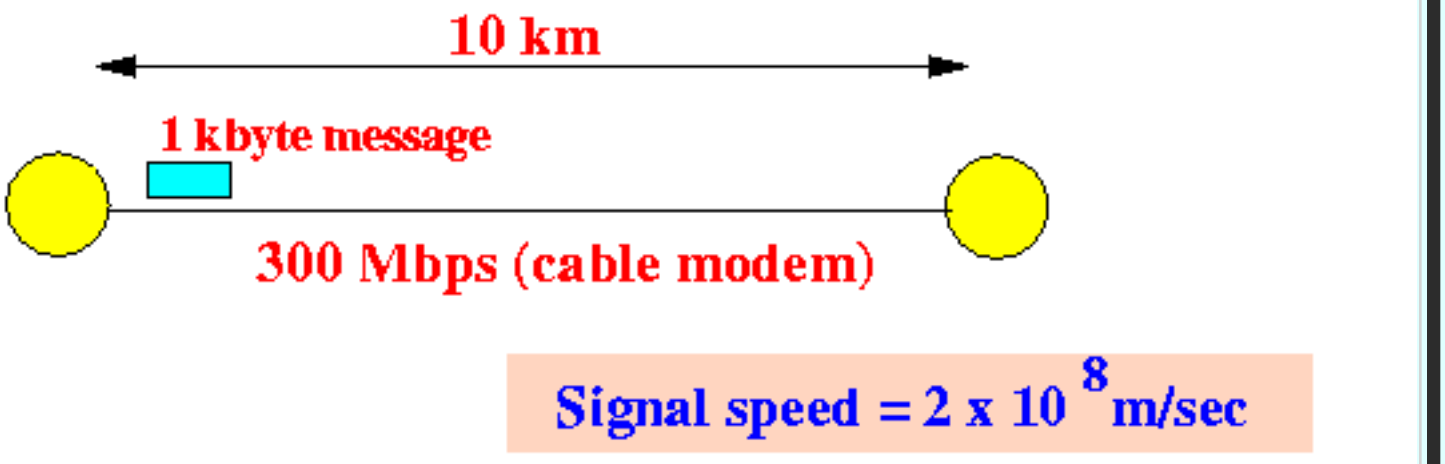
$$\text{Channel utilization} = \frac{0.16}{0.16 + 2 \times 0.00005}$$

$$= \frac{0.16}{0.1601}$$

= 99.94 %

- **Performance of Stop-and-Wait on high data rate (bandwidth) links**
  - **Case study 2:** efficiency of ***Stop-and-Wait*** on a ***high*** **speed** **transmission link**

    - **Now** find the **channel utilization** of **Stop-and-Wait** is the **data rate = 300 Mbps** (**everything else** remains **unchanged**)



Summary of the **operational parameters**:

- **Data rate** of the **transmission link** = **300 Mbps (kilobits/sec)**
- The **length** of the **link** = **10 km**
- **Sender** transmits **data frames** of size **1 kBytes**

- We will **ignore** the **time** needed to **transmit** an **ACK frame**

because: $t_{ACK} << t_{data}$

- **First**: compute t_{data}, t_{prop} and t_{ACK}.

- Compute t_{data}:

```
1 data frame  = 1 kbytes
              ~= 1000 bytes
               = 8,000 bits

Transmission speed  = 300 Mbps
                    ~= 300,000,000 bits in one sec

===> time to transmit 1 data frame = 8,000/300,000,000
                                   = 2.67×10^-5 sec


     t_data = 2.67×10^-5 sec
```

- Compute t_{ACK}:

- We ***assume*** that $t_{ACK} \sim= 0$

- Compute t_{prop}:

Signal speed = 2×10^8 m/sec
= 2×10^5 km/sec

Distance between sender and receiver = 10 km

===> time for signal to travel 10 km = $10 / (2 \times 10^5)$
= 5×10^{-5}

$t_{prop} = 5 \times 10^{-5}$ sec

- **Maximum channel utilization:**

$$\text{Channel utilization} = \frac{2.67 \times 10^{-5}}{2.67 \times 10^{-5} + 2 \times 5 \times 10^{-5}}$$

$$= \frac{2.67}{2.67 + 10} = \frac{2.67}{12.67}$$

$$= 21.07\ \%$$

In other words:

- **At most**:
  - **21%** of the **channel bandwidth**

  will be ***used*** to transmit **(*useful*) data frames**

- **Or:**
  - **at least *79%*** of the **channel bandwidth** will be ***wasted*** !!!

- **Verdict:**
  - **Stop-and-Wait** has ***good*** **channel utilization** for ***low*** **speed links**
  - It is **very *inefficient*** on ***high*** **speed links**

- **More *efficient* reliability protocols**
  - **Need**:

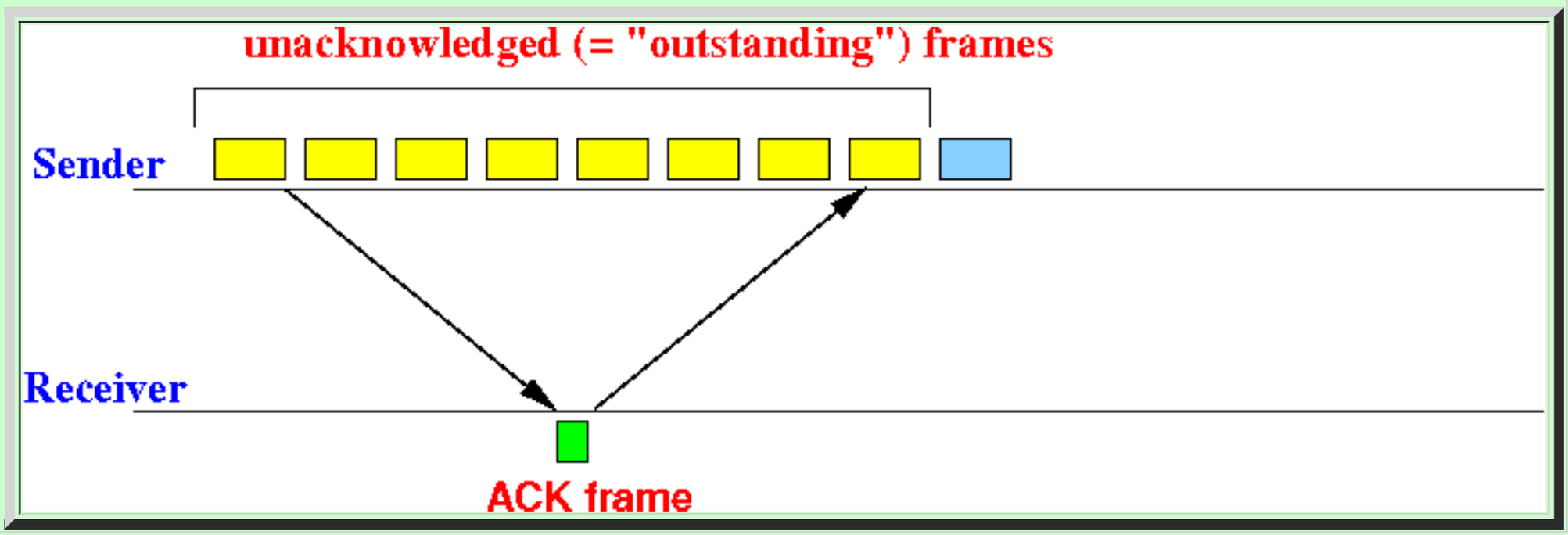
    - To achieve **better performance**, we **must** develop **more *efficient* (= complicated)** protocols.
    - We will next study the ***Sliding Window*** method.

Towards a more *more efficient* ARQ protocol

- **Performance problem in Stop-and-Wait protocol**
  - The ***cause*** of **poor performance** in the **Stop-and-Wait** protocol:
    - The **sender** must **wait** for an **ACK frame** before the **sender** can **transmit** the ***next* frame**

  - **Reason *why*** a **sender** must **wait**:
    - The **Stop-and-Wait protocol** will ***only*** need to **deal** with
      - ***One* lost data frame** or **ACK frame** at ***any* moment** in time.
    - This **simplifies** the **analysis** of its **correctness greatly**

- ***How to* improve the channel utilization of a communication protocol**
  - The **channel utilization** can be **increased** by
    - **allowing** the **sender** to transmit a ***multiply* of frames *without*** having to ***wait*** for an **ACK frame**:



  - **However**:
    - This **complicates** the **task** of
      - ***reliable* data communication**

      greatly !!!

- **The Sliding Window protocol**
  - **Sliding Window:**
    - **Sliding Window** = an **automatic repeat request (ARQ)** protocol, in which:
      - the **sending process** continues to send a **number of frames** specified by a **(send) window size** even **without receiving** an **acknowledgement (ACK) frame** from the **receiver**.

Wikipedia page: click here

- **The Go-back-N protocol**
  - **Go-back-N:**

    - **Go-Back-N ARQ** = a **specific instance** of the **Sliding Window** protocol, in which:
      - The **receive window size = 1**

    Wikipedia page: click here

A simple (but impractical) efficient reliable communication protocol

- **Reminder: reliable communication**
  - **Reliable communication means**:

    - **Each *data* frame** must be **delivered *once***

      - **Data frames** can be ***received multiple* times**,
      - **But *duplicate* frames** must be ***discarded***

    - Frames must be ***delivered*** to the **user** in the ***same* order** in which they were **sent**

- **The *simple* theorectical reliable communication protocol**
  - **Simplifying operational assumptions**:

    - ***Send* sequence number** in **data frame** has ***infinite* length**

      Consequently:

      - ***Each data* frame** has a ***unique* sequence number**

    - ***Recv* sequence number** in **ACK frame** has ***infinite* length**

      Consequently:

      - ***Each ACK* frame** can **identify** a ***data* frame *uniquely***.

    - ***Sender*** has ***infinite* buffer capacity**

      Consequently:

      - A **sender** can (and **will**) ***retain every data* frame** that it has ***transmitted*** (in its **buffer**)

      - When **necessary** (= when some **data frame** is **lost**), the **sender**

can **always able** to ***re-transmit*** the **data frame**

- ***Receiver*** has ***infinite*** **buffer capacity**

  Consequently:

  - A **receiver** can (and **will**) ***retain*** ***every*** ***data*** **frame** that it has ***received*** (in its **buffer**)

  - The **receiver** can use the ***unique*** **send seq. no** to **remove** ***all*** **duplicate data frames**

  - The **receiver** can use the ***unique*** **send seq. no** to **deliver** the **data frames** in the **transmission ordering**

- **How to** achieve **reliable data communication**:

  - **How to** ensure that ***each*** **data frame** is **delivered** ***once***:

    - The **sender** will **keep sending** a **data frame** that has **not** been **acknowledge**

      - The **sender** is **able** to do so **because** ***each*** **frame** is **saved** in the **sender's** ***buffer***

      - So ***each*** **data frame** will **eventually** be **received** by the **receiver**

    - The **receiver** uses the ***unique*** **send sequence number** to **remove** ***duplicates***

      Then:

      - ***Each*** **data frame** is **delivered exactly** ***once***

  - **How to** ensure that **data frame** is **delivered** in the ***same*** **order** as **transmitted**:

    Easy:

- The **receiver** delivers the **data frame** in ***increasing*** **send sequence number**

- **Making the "theoretical" protocol *practical***
  - **Fact:**

    - The **above protocol** **can** ensure ***reliable*** **communcation**

      (**I.e.:** it ***works***)

    The **only problem** is that it's **not practical** to **implement** the **protocol**....

  - It is ***impractical*** for the following **reasons**:

    - The **protocol** uses ***infinite*** **number** of **sequence numbers**

    - The **protocol** uses ***infinite*** **amount** of **buffer space**

    - The **sender** has ***infinite*** **buffer space** to store **transmission frames**

    - The **receiver** has ***infinite*** **buffer space** to store **received frames**

  - We will **deal** with the ***infinite*** **buffer space** issue **first**....

    We will **deal** with the ***infinite*** **sequence numbers** issue **later**.

Send buffer requirement and outstanding frames

- **The *use* of buffers in computer communications**
  - **Buffer:**
    - **Buffer** = **computer memory** used to **store data** (until the **data** is **no longer *needed***)
      - **Computers** have a **finite** amount of **buffers (memory)**

  - **Sender/Receiver buffer:**
    - **Send buffer** = **buffer** used by the **sender** to store ***data* frames**
    - **Receive buffer** = **buffer** used by the **receiver** to store ***data* frames**

  - **Note:**
    - There are **no buffers** used for ***ACK* frames** !!!
      - **ACK frames** are ***processed* immediately** when they are **received**
      - An **ACK frame** will **update** some ***status* information (= variables)** at the **sender**

- **Terminology**
  - ***Outstanding* frame:**
    - **Outstanding frame** = a ***data* frame** transmitted by the **sender** that has ***not*** been **acknowledged**

    Consequently:

    - The **sender** does ***not* know** whether an **outstanding frame**:
      - will be **(*correctly*) received** **or**
      - will be ***lost* (corrupted)**

- **The *buffering* constraint (requirement) on the sender**
  - The **buffering rule** that a **sender** must **obey**:
    - The **sender** ***must* keep *all* outstanding frames** in its **send buffer**

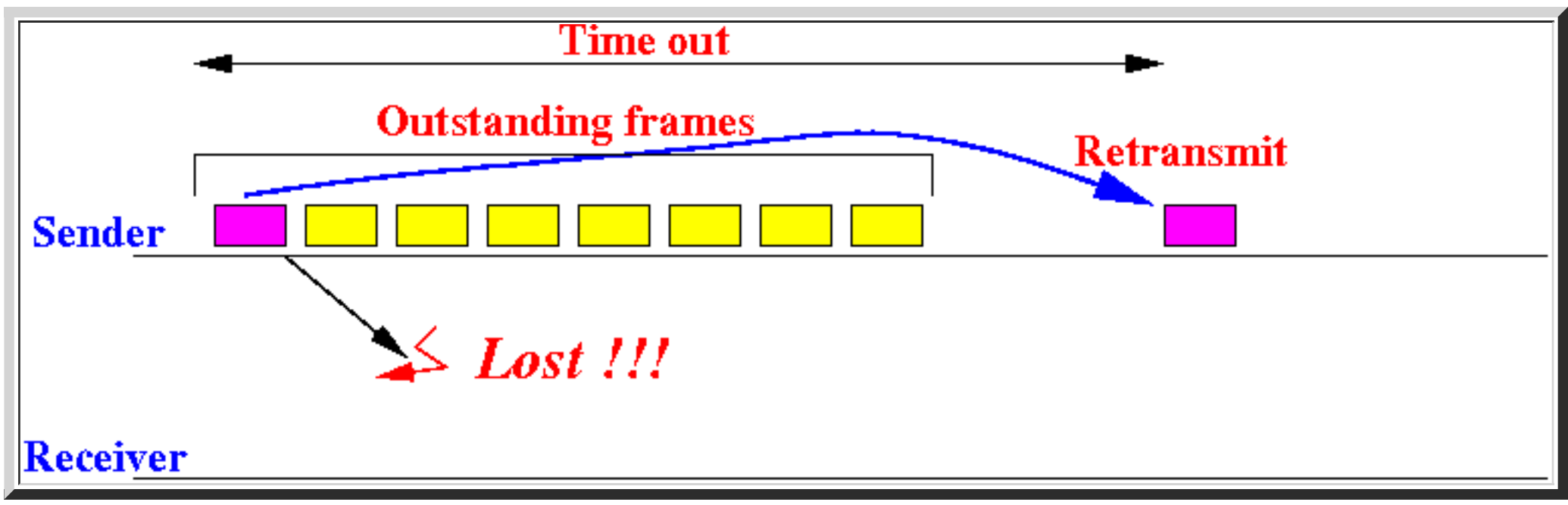
Reason:

- An ***outstanding*** **frame can** be **lost**

  Therefore:

  - The **sender must keep all outstanding frames** so it will be **able** to ***retransmit*** it when the **frame** is ***lost*** !!!

Graphically illustrated:

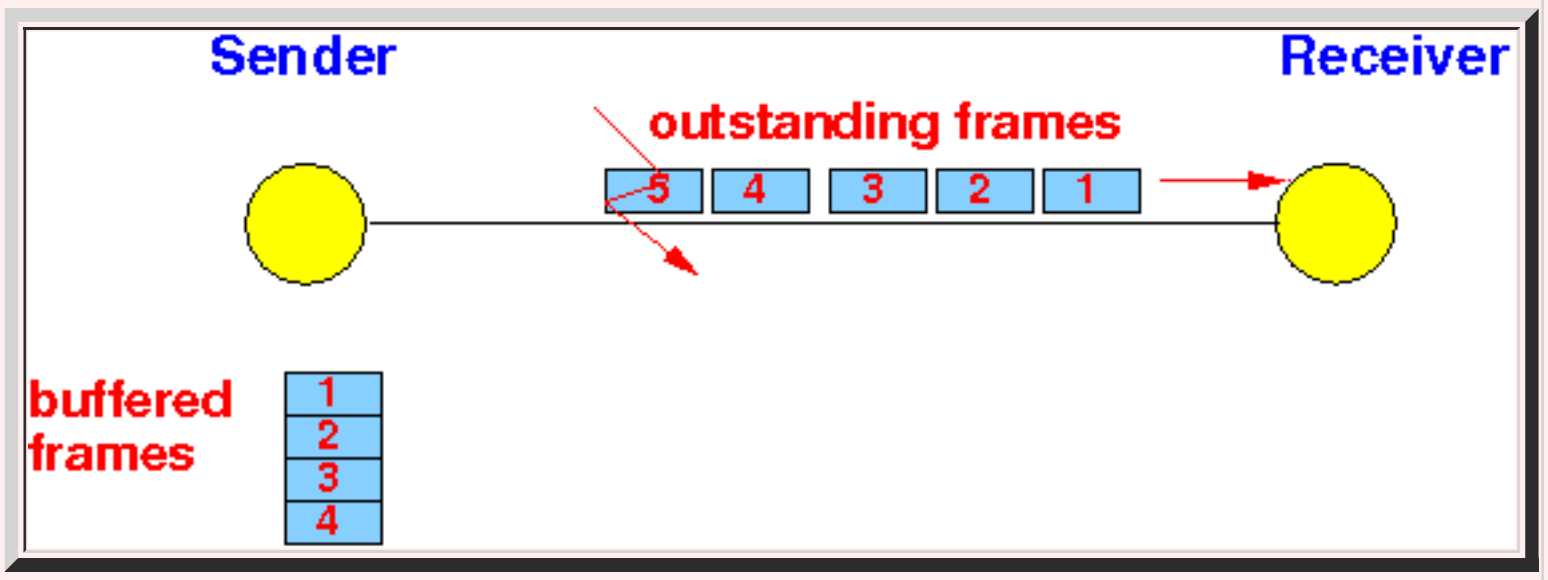


- **Maximum number of *outstanding* frames**
  - **Constraint** on **sender** to **ensure *reliable* communication**:

```
# outstanding frames  ≤  send buffer size
                         (# frames that send buffer can hold)
```

Reason to **impose** this **constraint**:

- **Suppose** the **sender** has ***k*** **buffers** and has ***k*+1** ***outstanding*** **data frames**:



Then: ***one*** of the ***outstanding*** **data frames** has **not** been **buffered** !!!

- In the **example**: **frame #5** has **not** been **buffered**

- If **that *unbuffered outstanding* frame** is **lost**:

- The **sender** ***cannot*** **retransmit** the ***lost*** **data frame** !!!

- **General transmission rule for sender**
  - To **achieve** the **best performance**, use:

```
# outstanding data frames = send buffer size
```

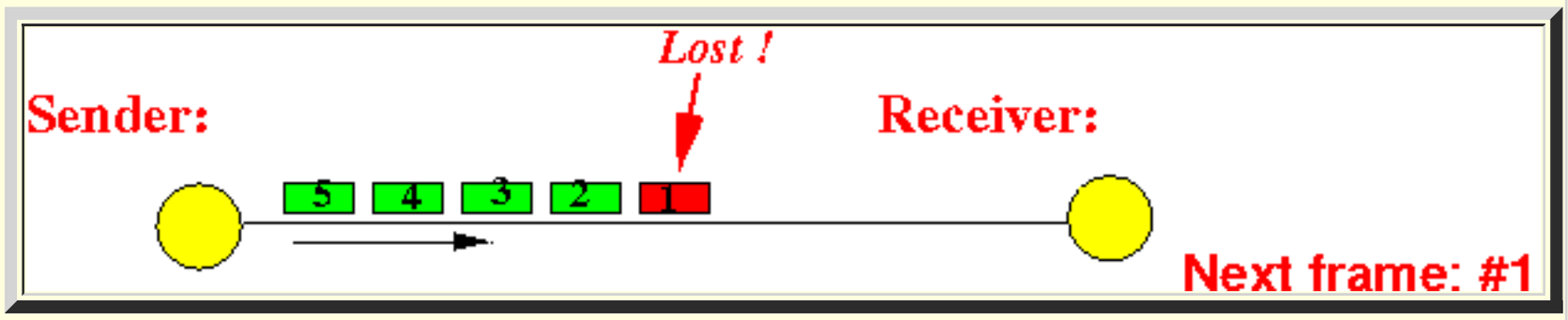
Receive buffer requirement and undeliverable frames

- **Undeliverable data frames**
  - **Undeliverable data frame:**
    - A **data frame** is ***undeliverable*** when ***some prior*** **frame** has **not** been **received**

- **The use of *receive buffers* (at the receiver)**
  - The **task** of the **receiver**:
    - **Deliver** the ***data*** **frames** from the **sender** in the ***order*** that was **transmitted**
  - The **purpose** of **receive buffers**:
    - **Receive buffers** = used to ***save*** **undeliverable** **data frames**

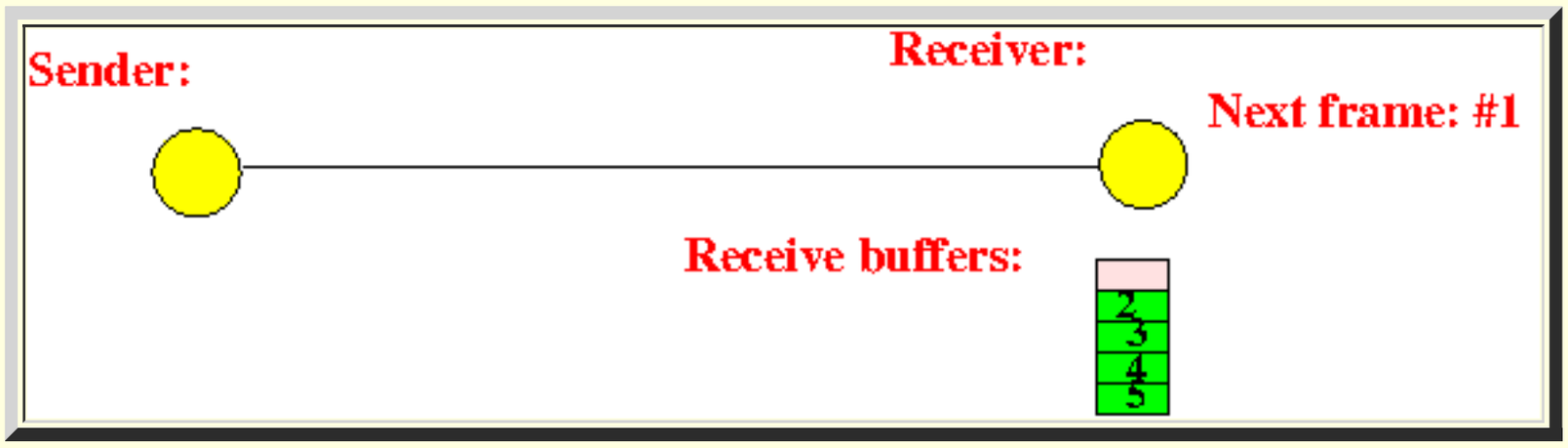
    Example:

    - **Sender** transmits **5 data frames**:

      Frame #1 is **lost**

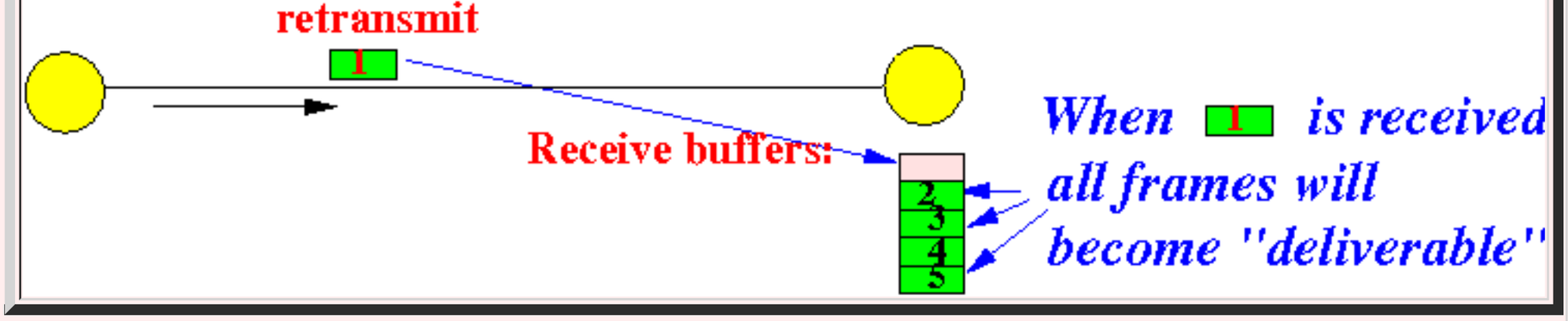
    - The **received data frames** are ***undeliverable***:

      The **undeliverable frames** are **stored** in the **receive buffer**

  - **Note:**
    - The **buffered** but ***undeliverable*** **frames** will **become** ***deliverable*** if:
      - The ***missing*** **earlier frame** is **received**

      Example:



When the **receiver** receives the ***retransmitted* frame 1**, the **receiver** can **deliver *all* received frames**

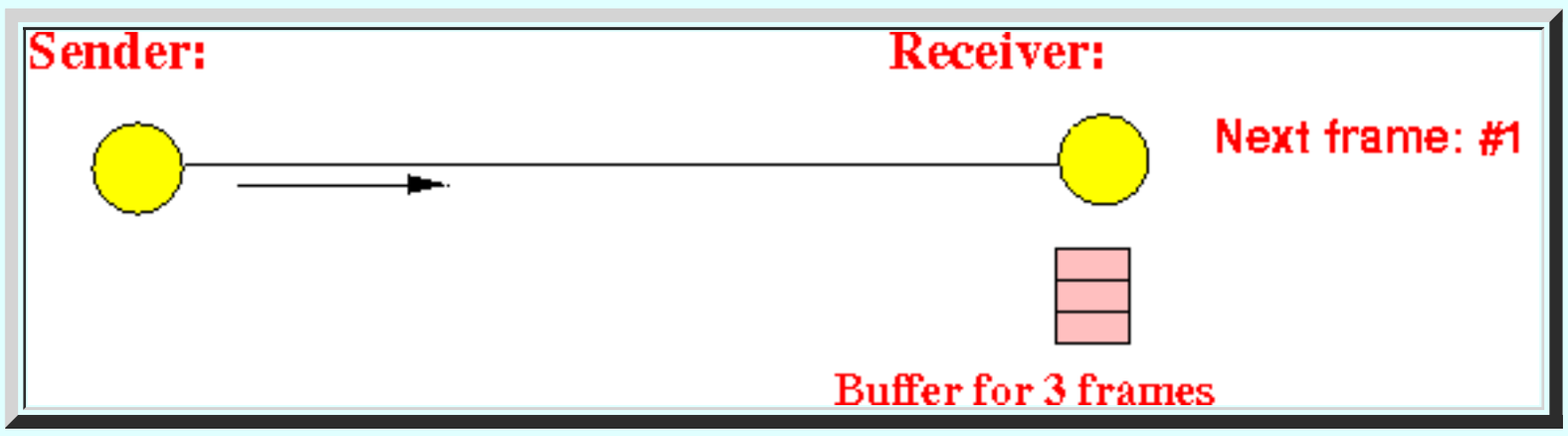
- **The effect of the receive buffer *size***
  - **Fact:**

    - **If receive buffer size = *k*, then**:

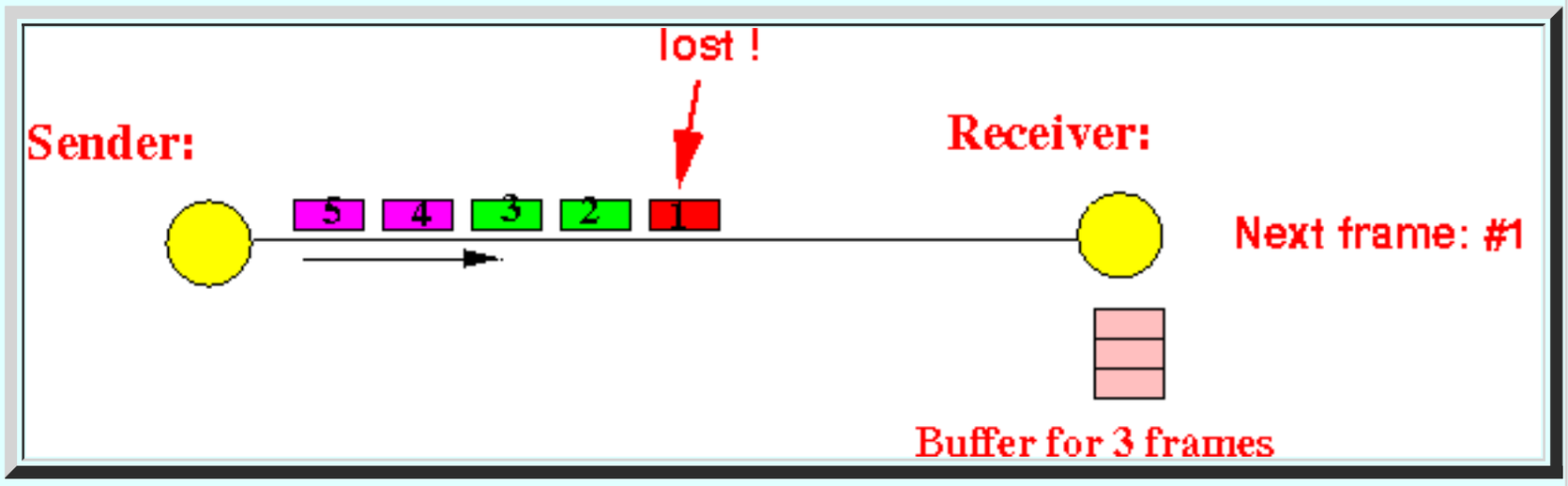
      - The **receiver** can **buffer** at most ***k* – 1 *undeliverable* data frames**

  - **Example:**

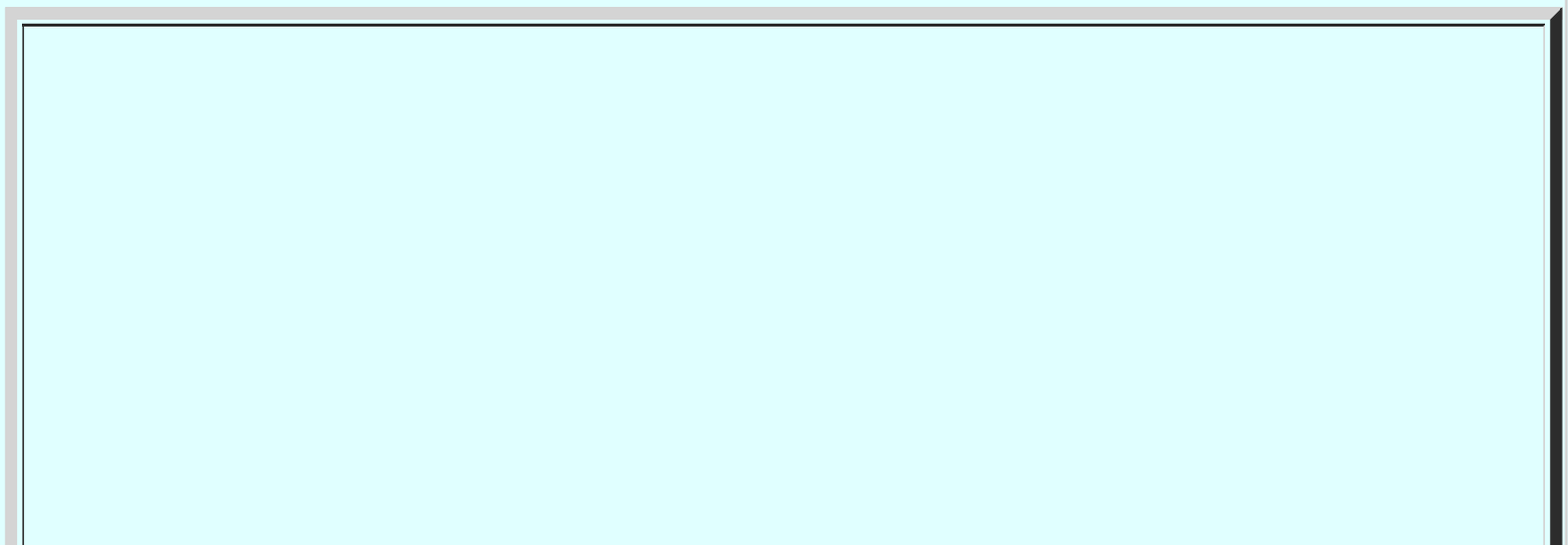
    - **Suppose: receiver** has reserved **buffers** for **3 frames**:

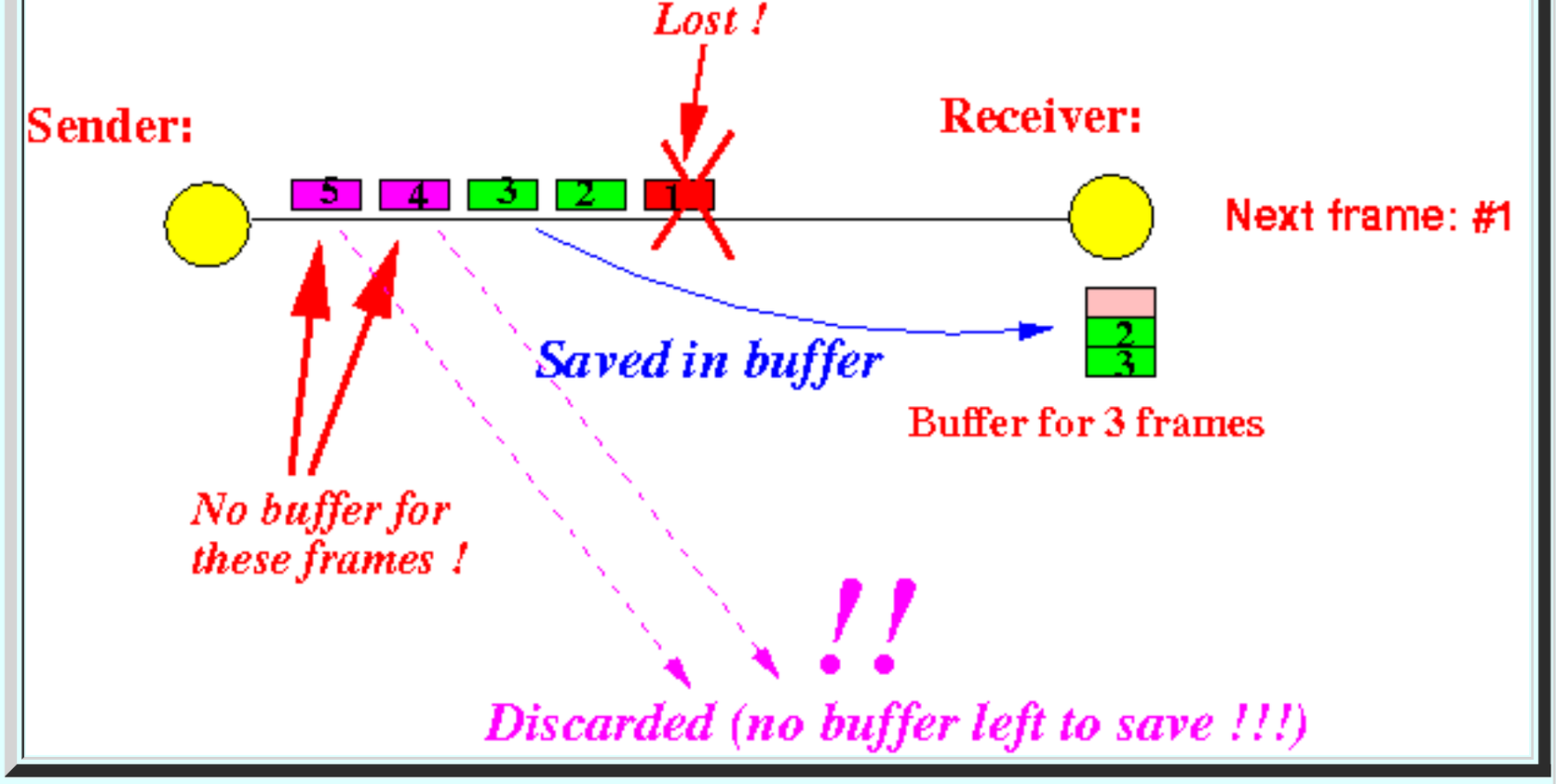


    - The **sender** transmits **5 frames** and the ***first* frame** is **lost**:



    - The **receiver** can ***only*** buffer **2 frames**:





Because:

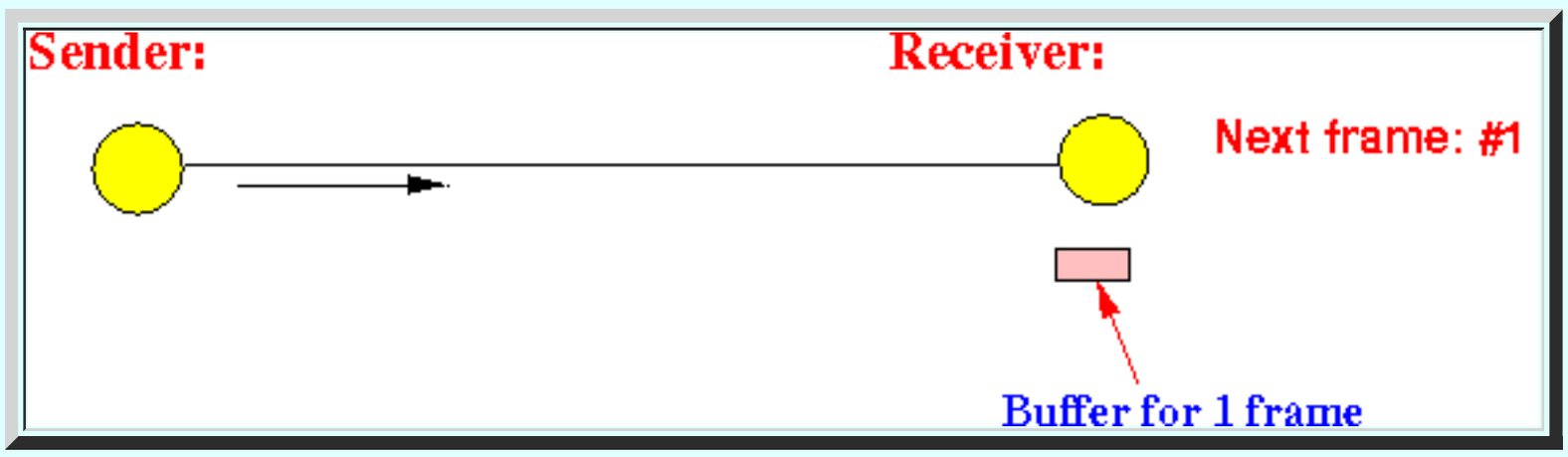
- **Receiver** must **reserve** a **buffer** for the ***missing*** **frame #1**
- **Receiver** can **buffer** the ***undeliverable*** **data frames 2, 3**
- **Receiver must *discard*** the ***undeliverable*** **data frames 4, 5**

(**In other words**: **frames 4 and 5** will be ***lost*** !!!)

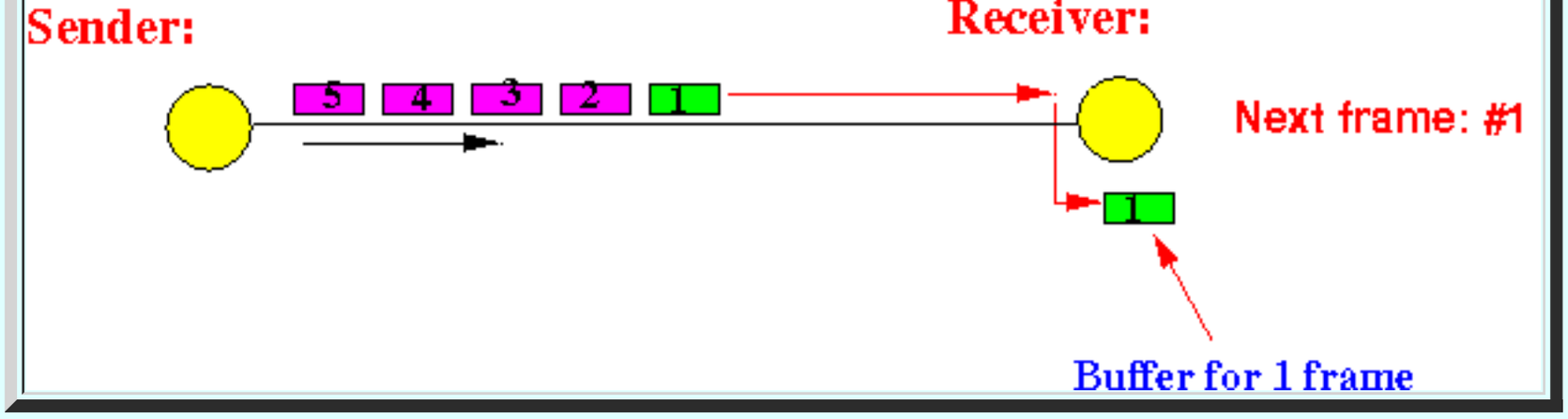
- **Minumum buffer requirement on a receiver**
  - **Fact:**
    - A **receiver** can **operate** ***correctly*** with ***1*** **receive buffer** !!!

      I.e.: minimum # receive buffers = 1 buffer !!!

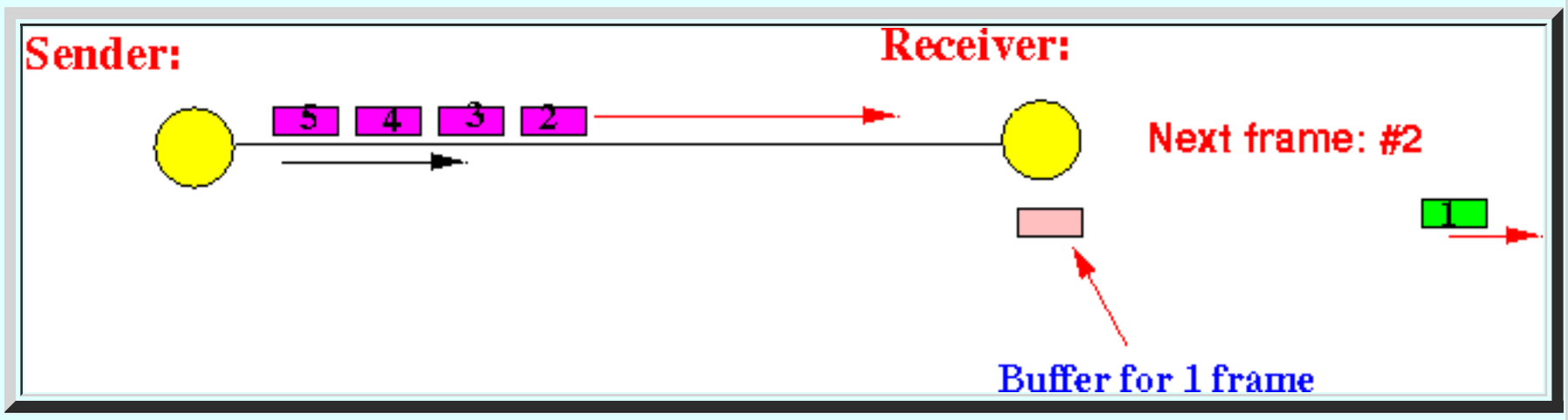
  - **Why** the **protocol** works **correctly** with ***only*** **1 receive buffer**:
    - **Suppose: receiver** has reserved **buffers** for **1 frame**:



    - ***If*** the ***first*** **data frame** is **received** ***correctly***:



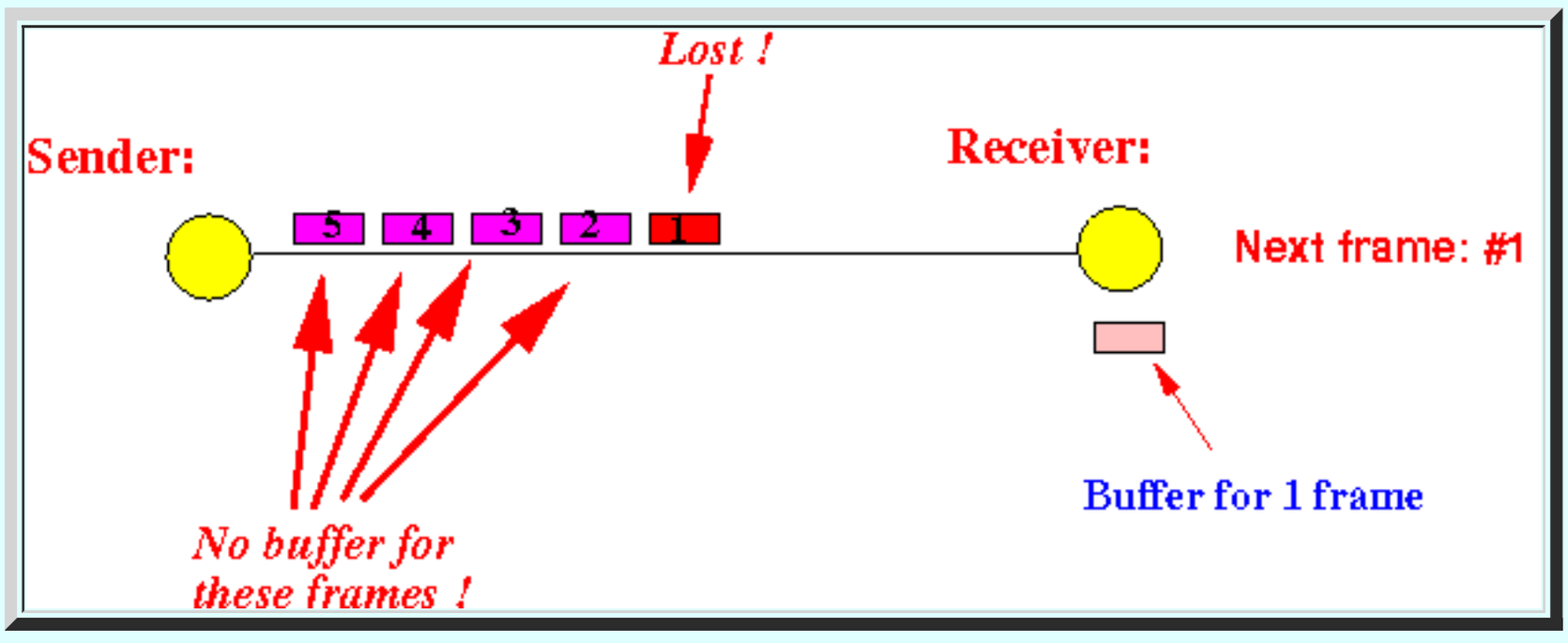
the **receiver** can ***deliver*** the **data frame**:



and **re-use** the **buffer** for the ***next*** **data frame**

- ***If*** the ***first*** **frame** is **lost**:

- The **receiver** will **discard** ***all*** the **undeliverable frames (2,3,4,5)**:



Because:

- **Receiver** must **reserve** a **buffer** for the ***missing*** **frame**

(The **effect** is the **same** as if frames **2, 3, 4 and 5** were ***lost*** ***also***)

- The **sender** can ***recover*** the ***data*** **frame losses** with ***retransmissions*** !!!

- **$64,000 question**
  - **$64,000 question:**

> - **What** is the **best setting** for **send buffer** and **receive buffer** ???

Relationship between Receive buffers and Send buffers

- **The relationship between send buffer size and receive buffer size**
  - **Rule** for **correctness**:

```
# outstanding data frame  ≤  send buffer size
```

because *otherwise*, the **sender** may **not** be able to **re-transmit** some ***lost* frame** (because ***that* data frame** was ***not* buffered**)

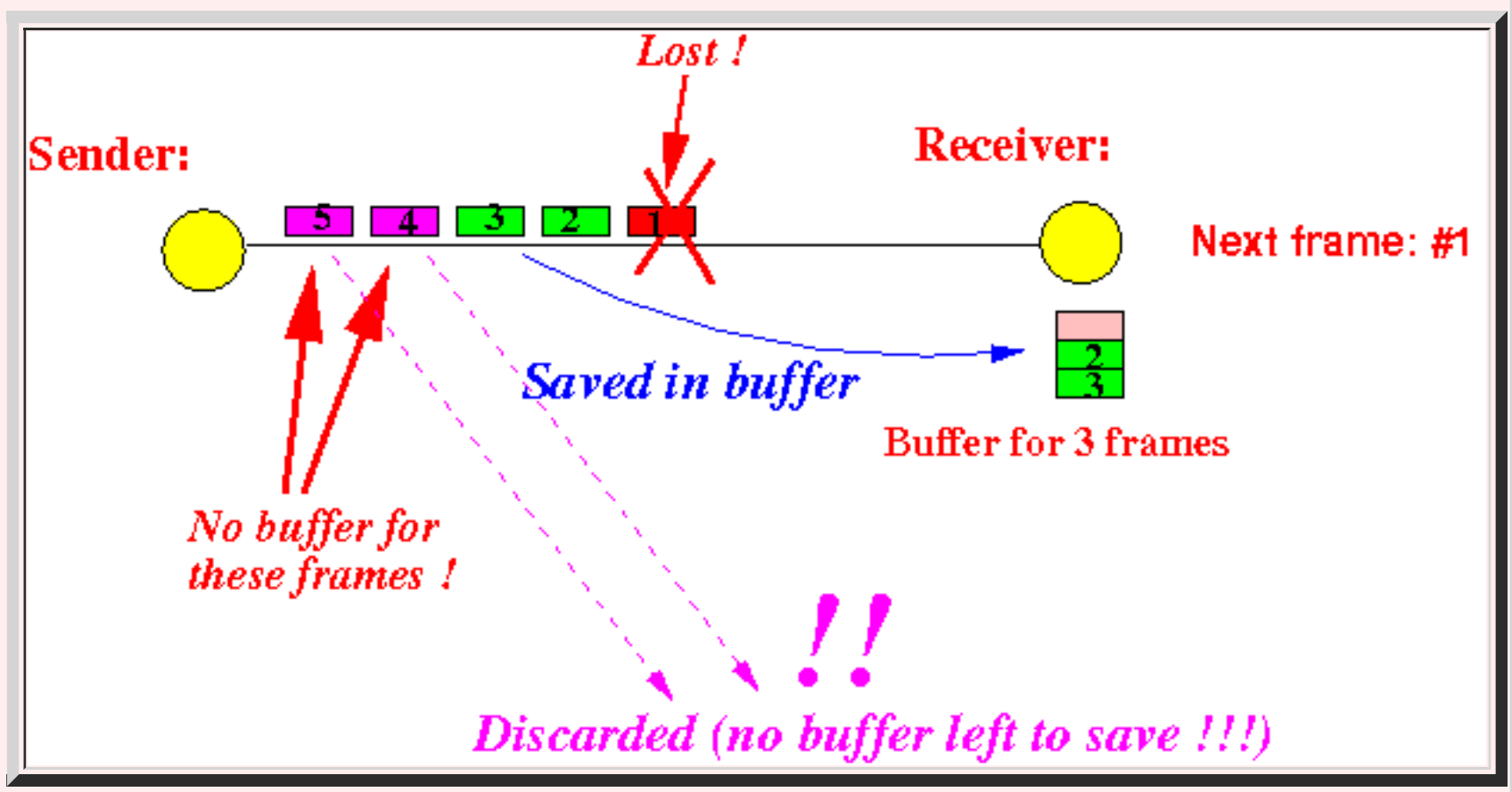
Best setting:

```
# outstanding data frame  =  send buffer size
```

  - **Rule** to achieve **better *efficiency*** (i.e.: **minimal *wastage***):

```
# outstanding data frames  ≤  receive buffer size

Or:

send buffer size  ≤  receive buffer size
```

  - **Reason:**
    - If `receive buffer size < # outstanding frames`, then some ***correctly* received data frames** can be ***discarded* (= lost)**:



Then:

- These **discarded frames must** be ***re-transmitted*** later !!

  ***Inefficient* use** of **transmission capacity** !!!

- **Setting used to achieve the best performance**
  - **Setting** used at the **sender** for **optimal efficiency**:

```
send buffer size = # outstanding frames
```

  - **Setting** used at the **receiver** for **optimal efficiency**:

```
receive buffer size = send buffer size
```

Selective Acknowledgement

- **Acknowledgement methods**
  - **Recall:**

    - An **acknowledgement frame** contains:

      1. **Indications** in the **frames** that the **frame** is an ***acknowledgement* frame** (and not **e.g.**, a ***data* frame**)

      2. A ***receive* sequence number**

         (The ***receive* sequence number differentiates** one **ACK frame** from ***another* ACK frame**.

         We **added** the ***receive* sequence number** to an **ACK frame** to **solve** the ***delayed* ACK error**: **click here**)

  - **Fact:**

    - An **acknowledgement** `ACK n` (`n` is the **receive sequence number**) can be ***interpreted*** in ***different* ways** !!!

  - ***Commonly* used meanings** assigned to `ACK n`:

    1. `ACK n` **acknowlegdes *only*** the **frame** with **sequence number** `n`

    2. `ACK n` **acknowlegdes *all*** the **frame** with **sequence number** $\leq$ `n-1`

- **Selective Acknowledgement**
  - **Selective Acknowledgement:**

    - **Selective Acknowledgement** = an **acknowlegdement *protocol* (= agreement)** where:

- The frame **ACK *n*** acknowledges ***only*** the ***data* frame *n***

- **Comment:**

  - That's what I have been **using** all the time ***without*** telling you the ***official* name** of the **protocol**....

Window

- **Window**
  - A **window** (in **computer networks**) is:

    - **Window** = a ***consecutive*** **sequence** of **integer values** used as ***sequence*** **numbers** for **frames**

      Examples: 2 windows

      - $W_1 = 0, 1, 2, 3, 4, 5$
      - $W_2 = 3, 4, 5, 6$

    - **Window size** = the **number** of **integers** in the **window**

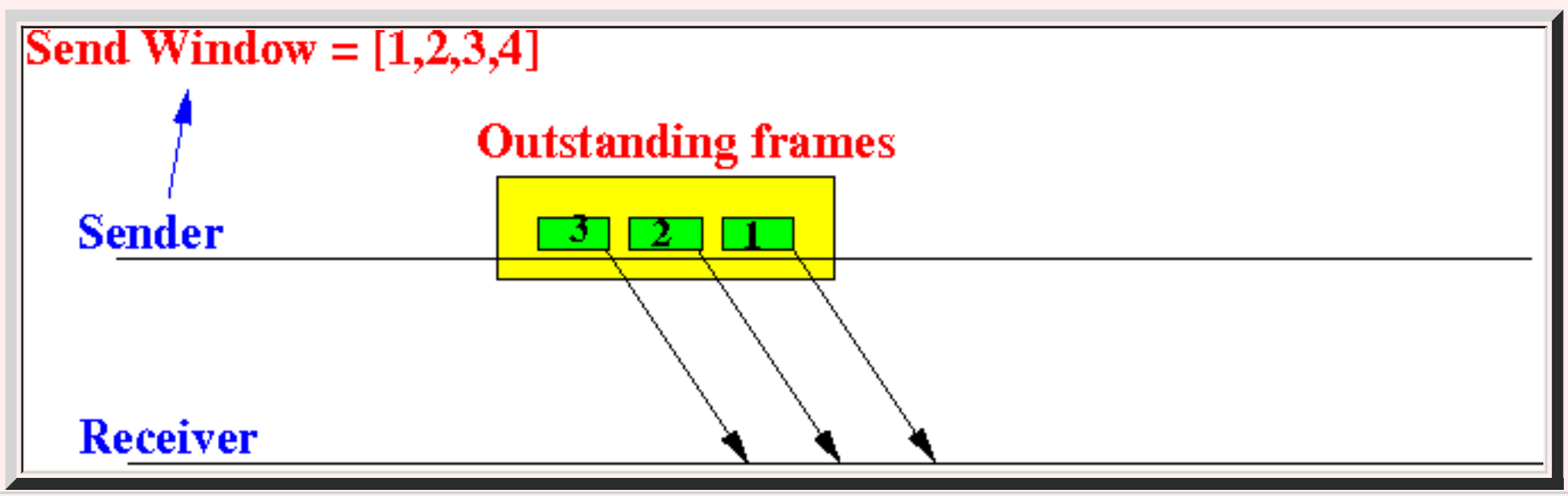
      Examples:

      - **Size** of $W_1 = 6$
      - **Size** of $W_2 = 4$

- **Sender and receiver windows**
  - **Send/Sender window**:

    - ***Send*** **Window** = the **list** of **sequence numbers** that the **sender** can **use** to **label (= identify)** an ***outstanding*** **frames**

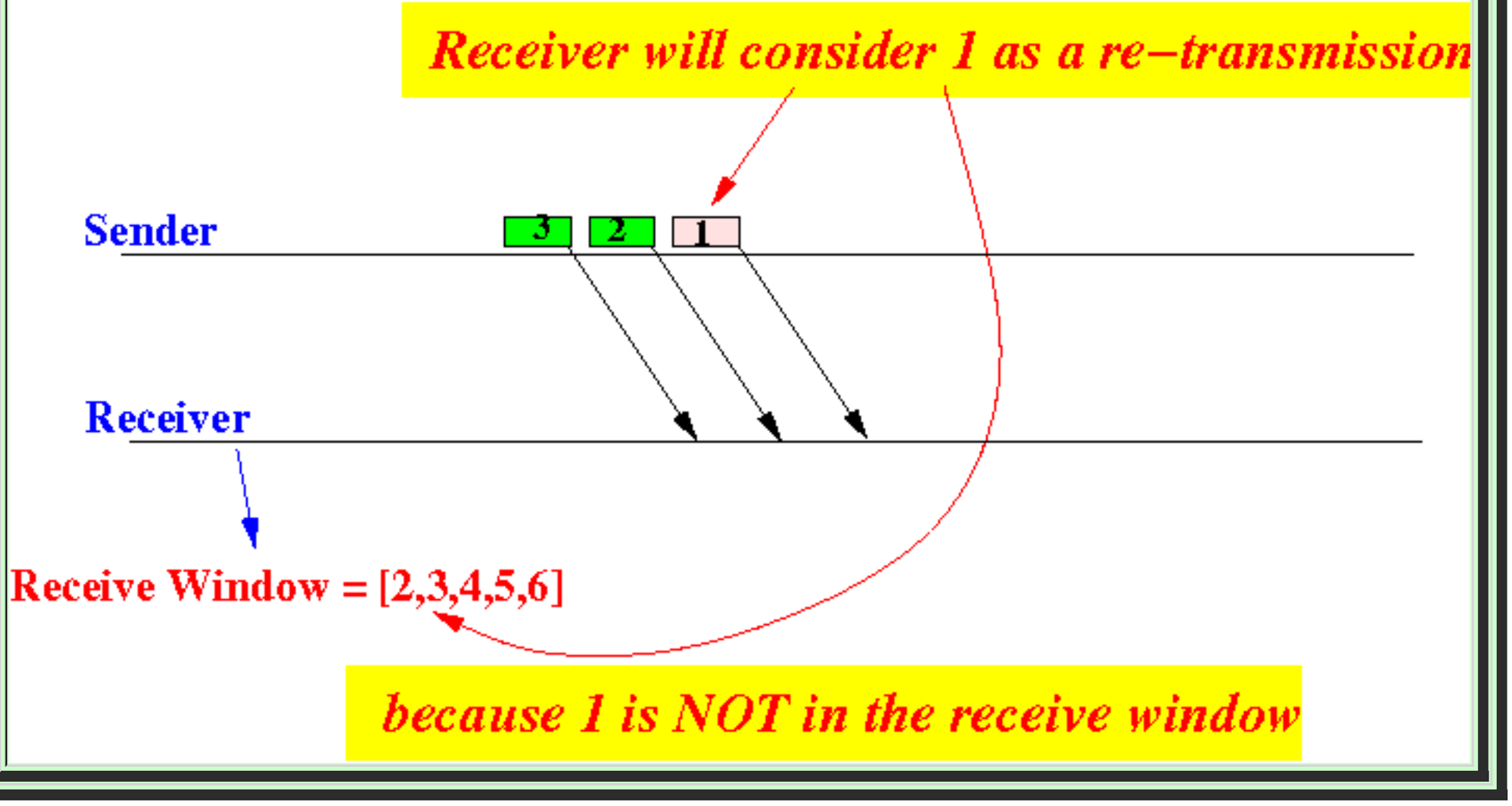
      Example:



  - **Receive/Receiver window**:

    - ***Receive*** **Window** = the **list** of **sequence numbers** that the **receiver** will **consider (= accept)** as ***new*** **(data) frames**

      Example:



- **Important facts:**

  - **Sender window** and **receiver window** can be **different** in ***size*** **(= width)**

    - The ***best*** **setting** is **send window size = receive window size**

    - But this ***setting*** is ***not*** **required** !!!

  - **Sender window** and **receiver window** are (**must**) ***lined up*** at the ***start*** of the **communication session**

    Lined up means:

    - ***first*** **value** in **sender window** = ***first*** **value** in **receiver window**

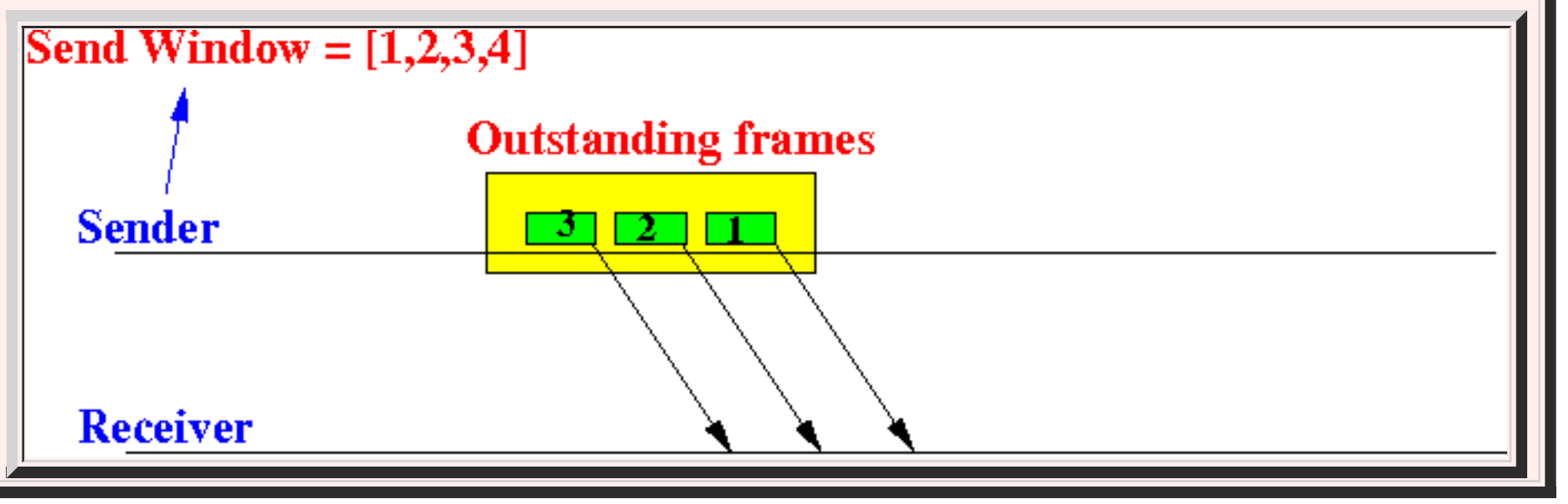
    Example:

The *Sender* Window

- **Send/Sender window**
  - **Send/Sender window:**

    - ***Send*** **Window** = the **list** of **sequence numbers** that the **sender** can **use** to **label (= identify)** an ***outstanding*** **frames**

      Example:



  - **Note:**

    - ***# outstanding*** **frame** of the **sender** **≤** **send window size** !!!

- **Updating the send/sender window (= "slides")**
  - **Reminder:**

    - **ACK *n* acknowledges *only*** the **data frame *n***

  - **Update algorithm** for the **sender window:**

```
Let Sender Window = [ A .. B ]

Sender receives an ACK X frame
(I.e.: X = recv seq no in received ACK frame)

/* ================================================
   X = the number in the ACK X frame
   ================================================ */
if (  X  ==  A )
{
    /* ================================================
       "Slide" the window forward
       ================================================ */
   i = A;

   while ( i ≤ B ) do
   {
      if ( frame i has been acknowledged )
```

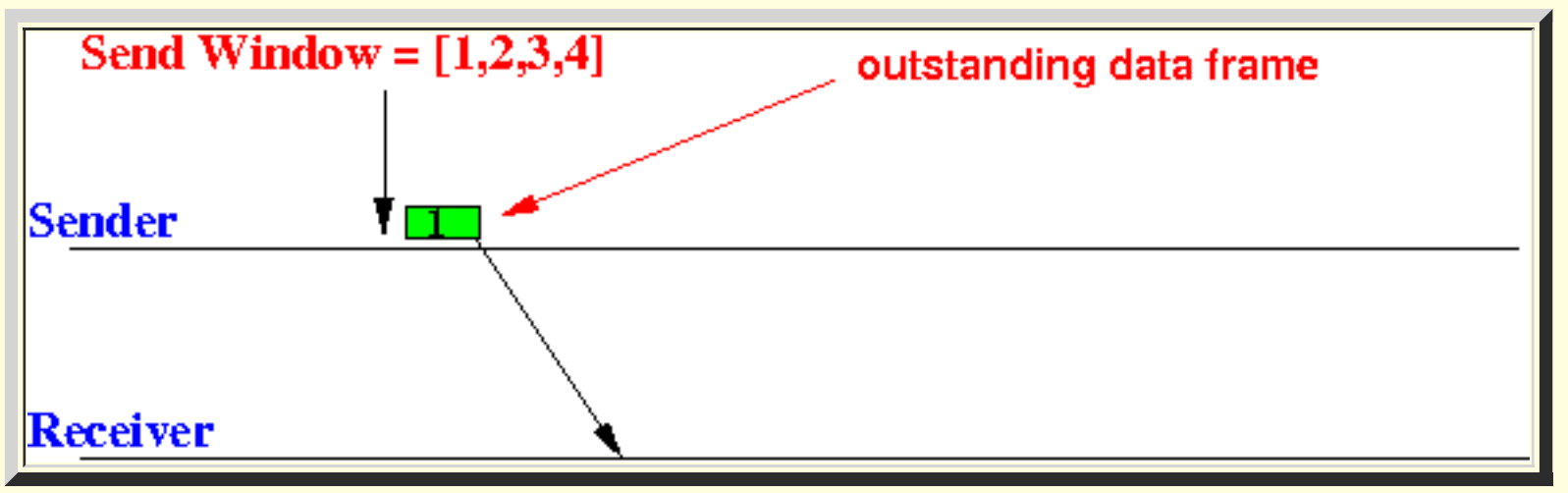
```
        {
            Release buffer for frame i

            i++;                // Check next frame in send window
        }
        else
        {
           break;         // Stops at the first unacknowledge frame
        }
    }

    Sender Window = [i, i+1, .., i+(B-A)+1)]
                        // Update (slide) sender window to start at [i .....]
}
```

- **Example 1:**

  - Suppose the **current *sender* window** is:

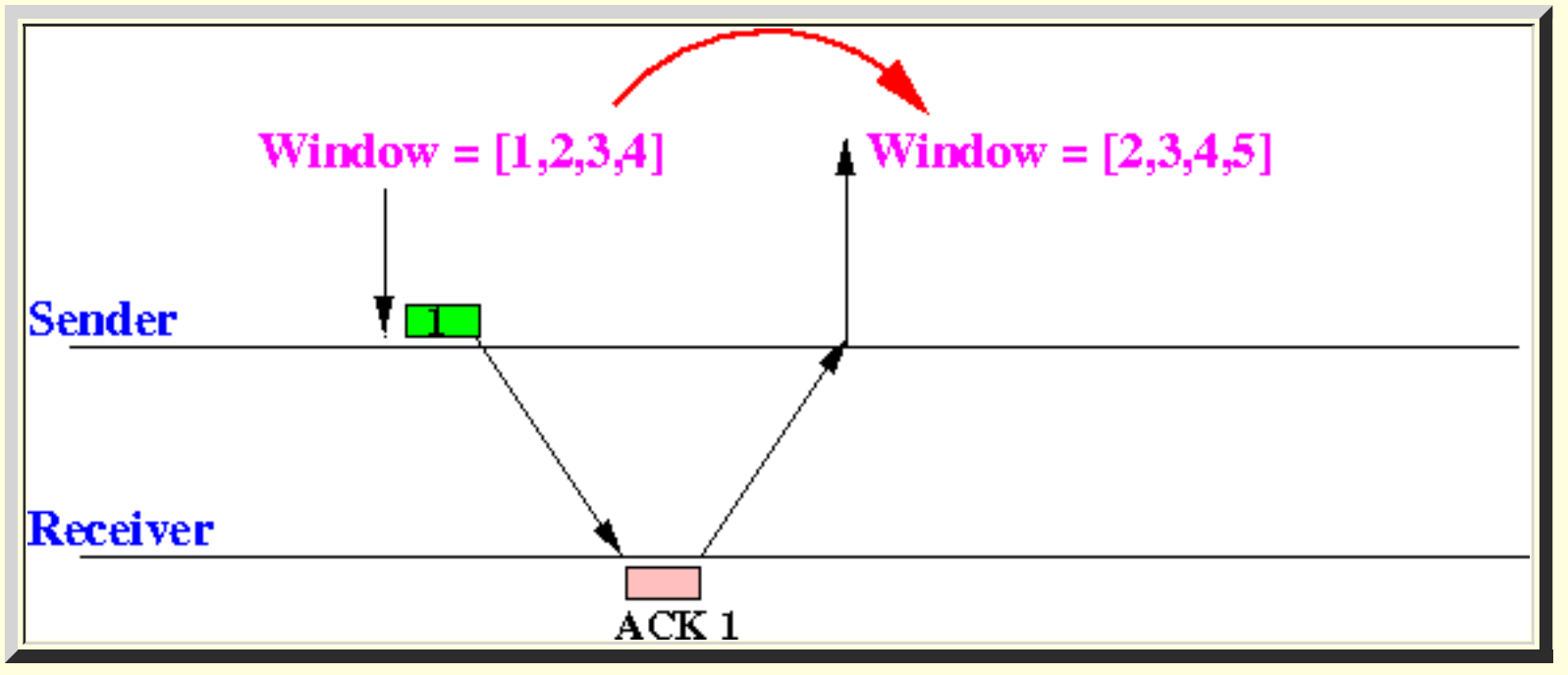
    ```
    Sender window = 1, 2, 3, 4
    ```

  - The **sender** transmits the **next data frame**:

    The ***next*** **data frame** will use the ***first*** available **number** in the **sender window** is its **send seq. no.**!!!

  - When the **sender receives the ACK frame, the send window will be updated to [2,3,4,5]:**

- **Example Program:** (Demo above code)

  Example

  - Prog file: **(cd /home/cs455001/demo/SelAck; run)**

  How to run the program:

- Send **1 new frames**

- **Example 2:**

  - **Suppose:**

    - **Sender window** = **[1,2,3,4]**

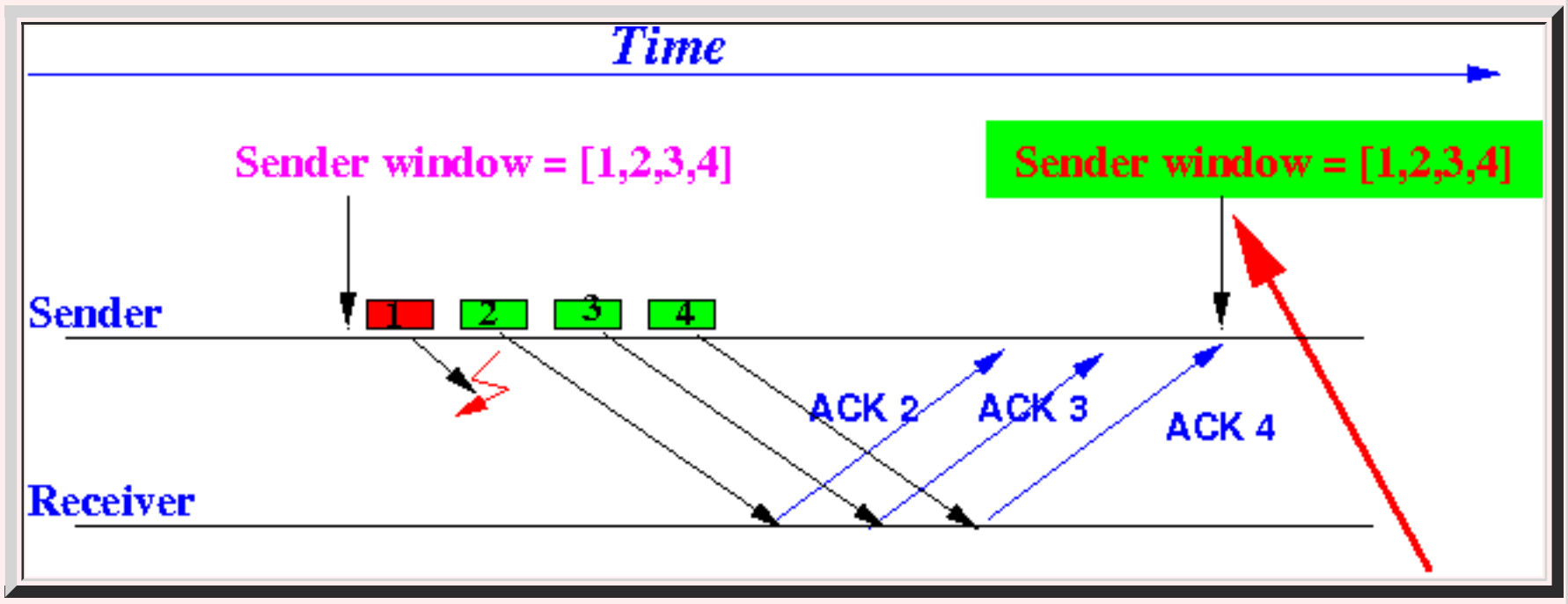
    Events:

    - The **sender** transmits **frames 1,2,3,4**
    - **But**: **Frame 1** is **lost**
    - The **receiver** transmits back **ACK frames 2,3,4**

    How the **sender** process the ***received*** **ACK frames**:

```
Sender window = [1 .. 3]

Received ACK 2:

       Test    2 == 1  ?   No - Done
                           (do not update sender window)

Received ACK 3:

       Test    3 == 1  ?   No - Done
                           (do not update sender window)

Received ACK 4:

       Test    4 == 1  ?   No - Done
                           (do not update sender window)
```

Graphically:



Note:

- The **sender window** does ***not*** **slide** forward !!!

  (because ***no*** **send buffer** has been ***released*** !!!).

- **Example 3:** **previous example** ***continues***

- The **sender** **times out** and **retransmits** **frame 1** and **frame 1** was **received** ***correctly***

- **Current sender/receiver windows:**

  - **Sender window** = **[1,2,3,4]**

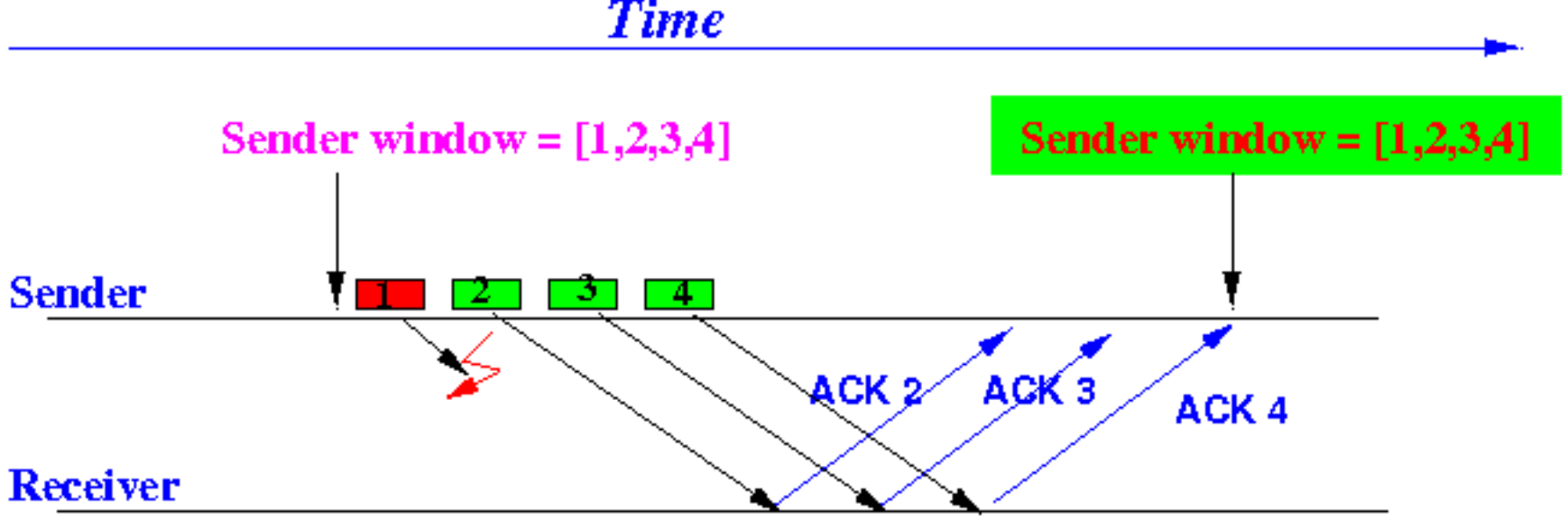
  Events:

  - The **sender** retransmits **frame 1**
  - **Frame 1** is **received**
  - The **receiver** transmits **ACK frame 1**

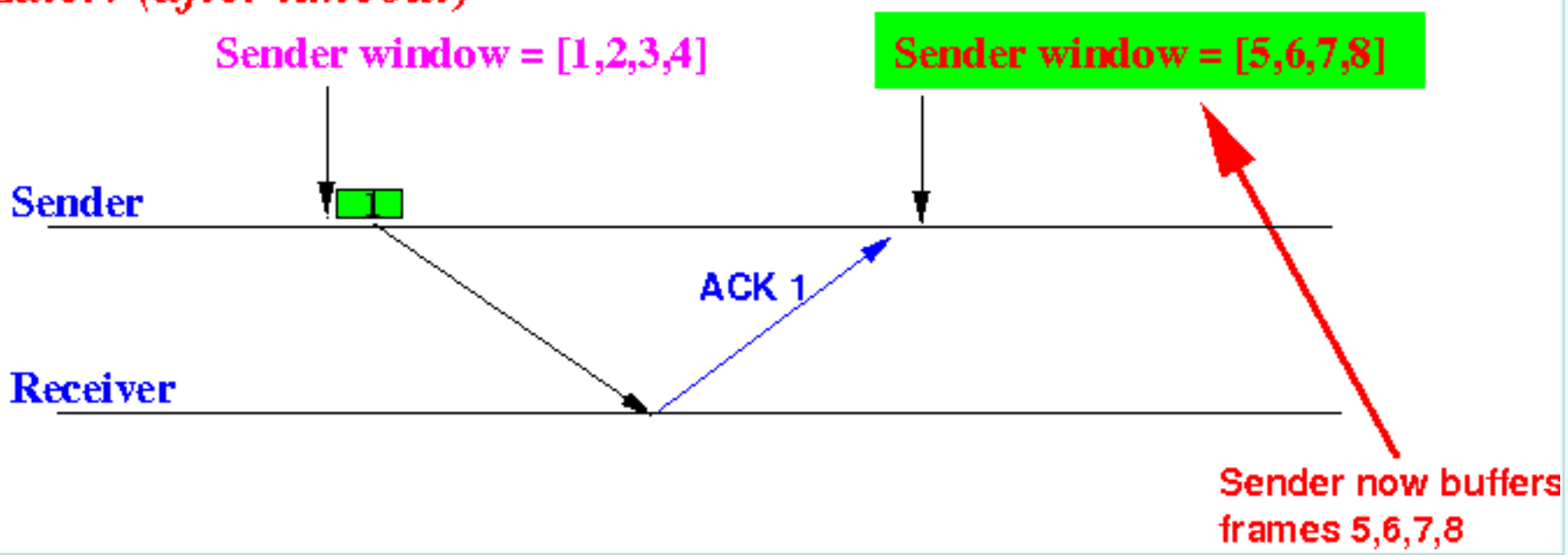
  How the **sender** process the ***received*** **ACK frame (#1)**:

```
Send window = [1 .. 3]

ACK 1:     1 == 1 ?    Yes

       for each frame Y = 1, 2, 3, 4 do:

           Y = 1:    Y is ACKed ?   Yes
                                    ===> Release buffer for frame 1
                                         Sender Window = [2 .. 5]

           Y = 2:    Y is ACKed ?   Yes
                                    ===> Release buffer for frame 2
                                         Sender Window = [3 .. 6]

           Y = 3:    Y is ACKed ?   Yes
                                    ===> Release buffer for frame 3
                                         Sender Window = [4 .. 7]

           Y = 4:    Y is ACKed ?   Yes
                                    ===> Release buffer for frame 4
                                         Sender Window = [5 .. 8]
```

Graphically:



Later: (after timeout)



Result:

- The **sender** will ***release*** **all 4 buffered frames (1,2,3,4)**
  - **Now** the **sender window** will **slide** forward by **4 positions**

  The **sender** will **now** buffer the **data frames 5,6,7,8**

- **Example Program:** (Demo above code)

  Example

  - Prog file: **/home/cs455000/demo/SelAck/run**

  How to run the program:

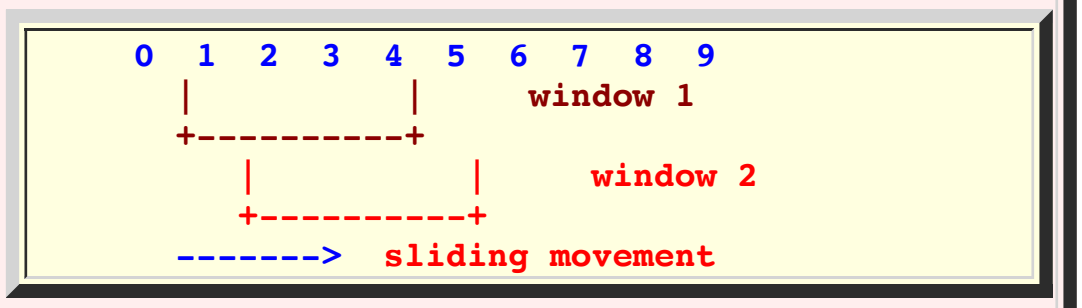
  - Send **2 new frames**
  - **Pause**
  - Click on **frame 0** and **kill it**
  - **Resume**

    Observe that:

    - The **Sender window** will **slide** only when the **retransmission** was **received** and **ACKed**

- **"Sliding" window**
  - The **movement** of the **sender window** is called:

- **"Sliding" window:**



- The **reliability protocol** that uses **sender/receiver windows** are called:

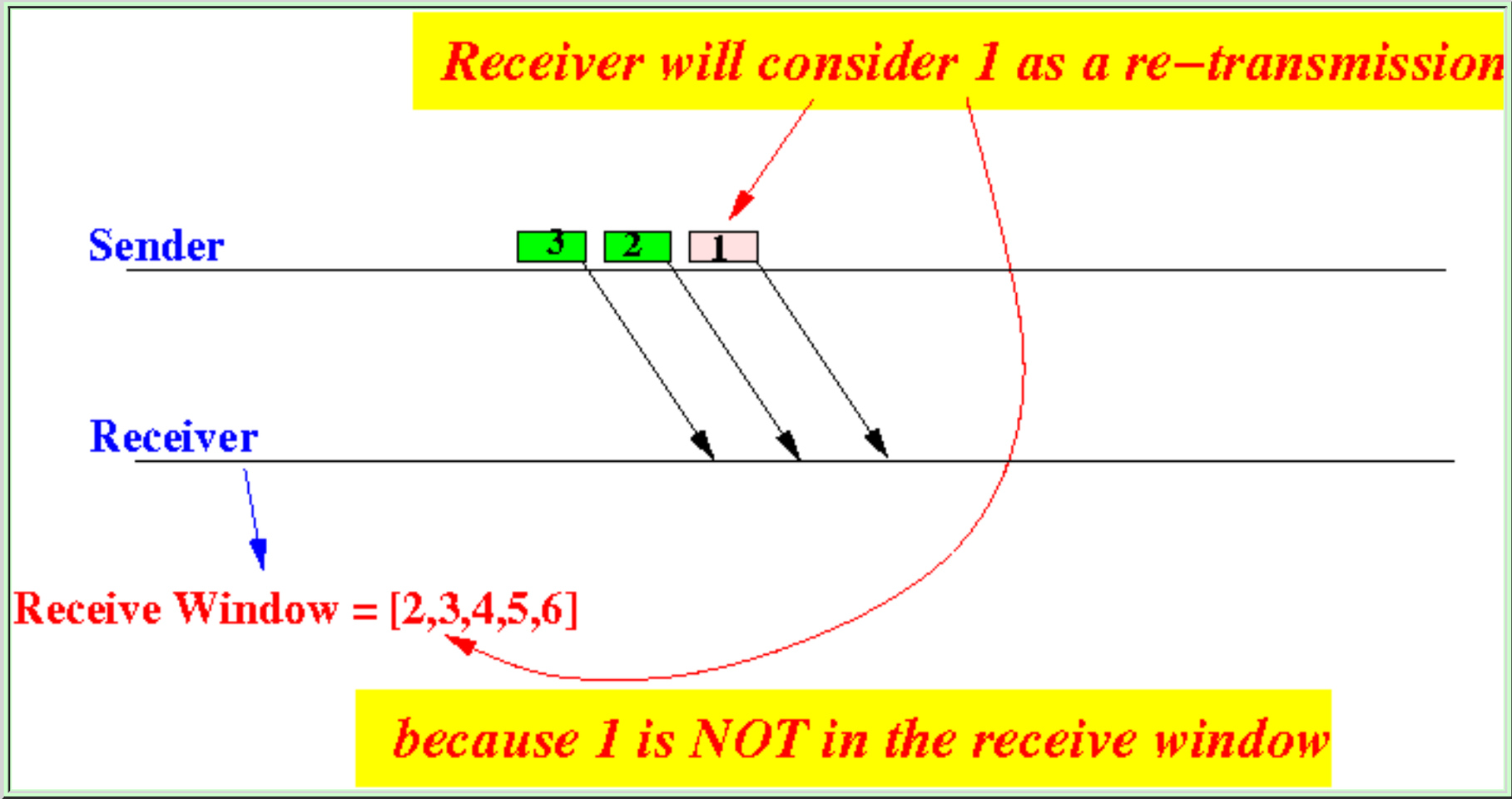
  - **Sliding window** schemes

The *Receiver* Windew

- **Receiver window and Receiver buffer**
  - **Recall**: **Receive/Receiver window**

    - ***Receive* Window** = the **list** of **sequence numbers** that the **receiver** will **consider (= accept)** as ***new* (data) frames**

      Example:



  - **Recall**: **Receive/Receiver buffer**

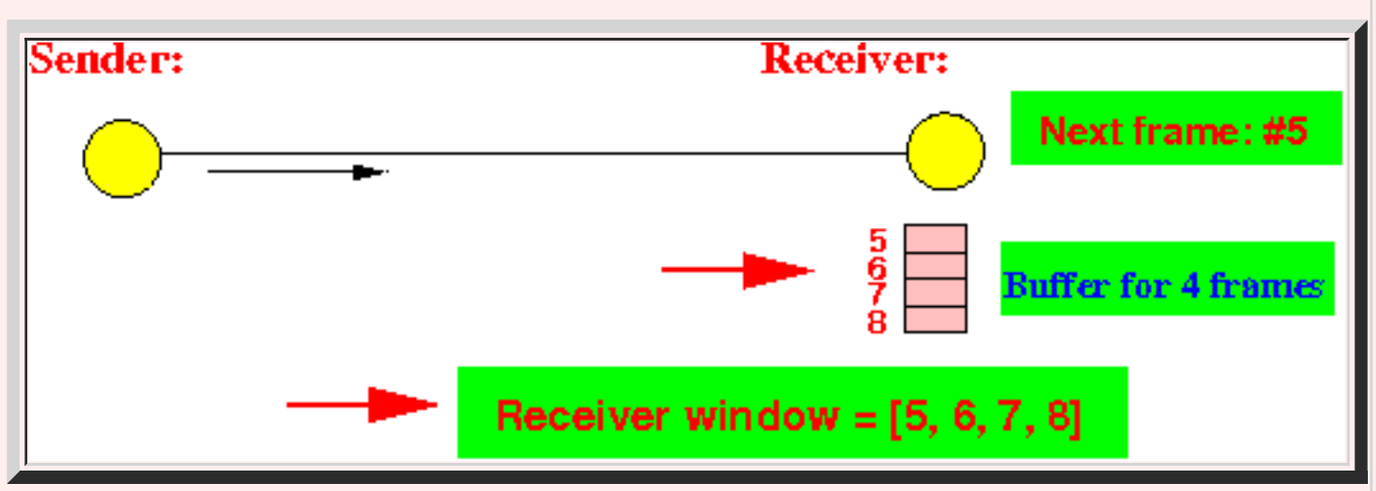
    - **Receiver buffer** = **buffer** in the **receiver** to store ***undeliverable* data frames**

    Example:

    - **Receiver window** = **[5,6,7,8]**

      Receiver has **4 (frame) buffers**

      Then:



      Receiver will **buffer** frames **5,6,7,8**

    (**Typically: receiver window size = receive buffer size**)

- **How the receiver window changes (= "moves")**
  - **Update algorithm** for the **receiver window**:

```
Let Receiver Window = [ A .. B ]

Receiver receives a data frame;

Let X = send sequence number in received data frame;
```

```
if ( A  ≤  X  ≤  B )
{
   Save the received data frame in the receiver buffer;

   Send ACK X;

    /* ---------------------------------------
       Try to deliver data frames in sequence
       --------------------------------------- */
    if ( X == A )
    { /* ------------------------------------------------------
         First new frame is received, we can
         deliver some data frames
         ------------------------------------------------------ */

       for ( frame i = A;  i ≤ B;  i++ ) do
       {
          if ( frame i was received (previously) )
          {
             Deliver frame i
          }
          else
          {
             break;
          }
       }

        Receiver Window = [i  ..  i+(B-A)+1]
                              // Advance receiver window !!!
    }
}
else
{  /* ============================================
      Frame X has bee received previously !!!
      ============================================ */
   Discard frame !!!

   Send ACK X;     // The ACK for frame X may have been lost !!!
                   // That may be why the sender re-transmits X !!!
}
```

- **Example 1:**

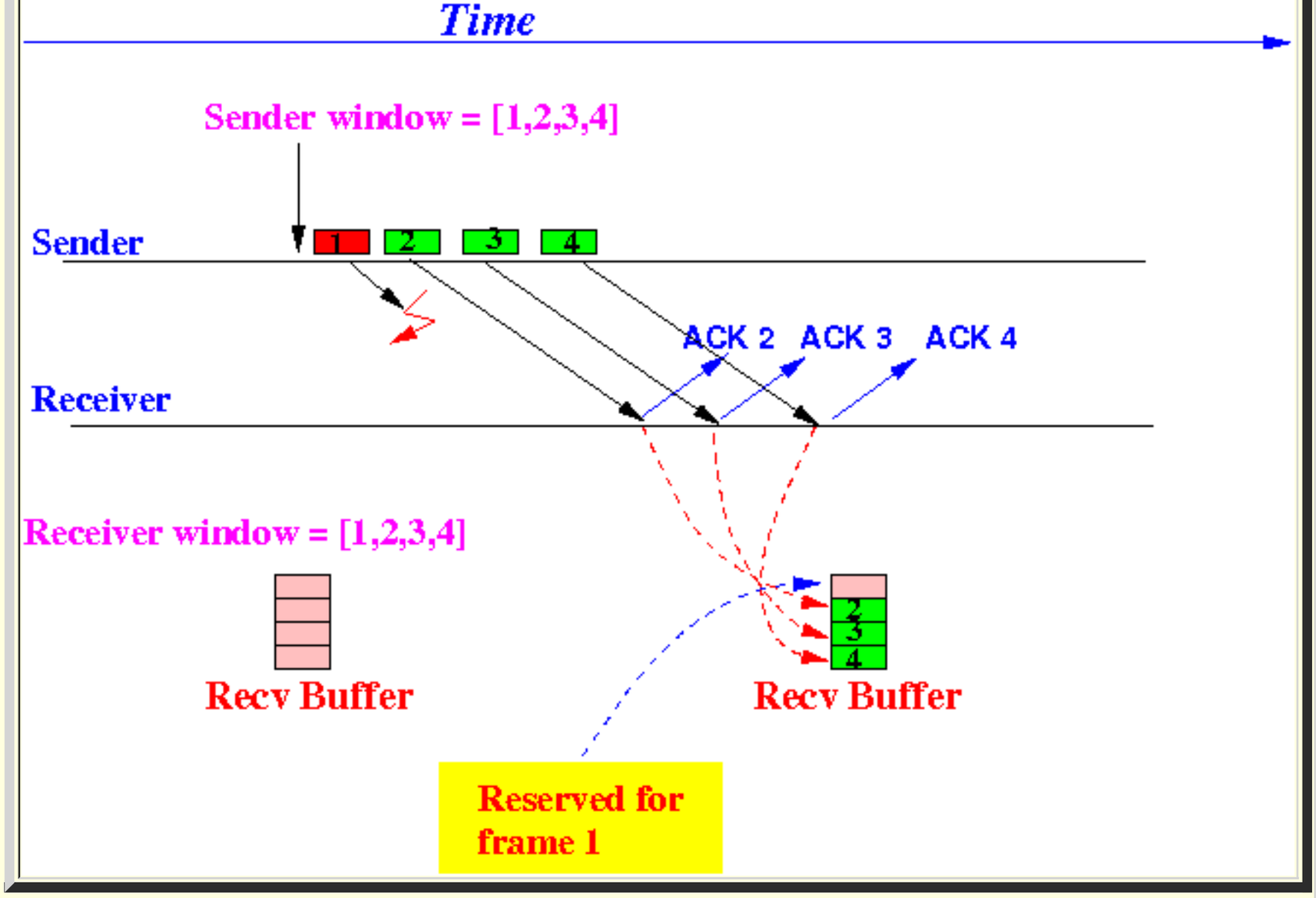
- **Suppose:**

  - **Sender window** = **[1,2,3,4]**
  - **Receiver window** = **[1,2,3,4]**

  Events:

  - The **sender** transmits **frames 1,2,3,4**
  - **But**: **Frame 1** is **lost**

  Graphically:



Explanation: how the **receiver** process the ***received*** **frames**:

```
Receiver window = [1 .. 4]

Frame 1:    lost

Frame 2:
            1 ≤ 2 ≤ 4     ?   Yes   ===> buffer frame 2
                                        Send ACK 2

            2 == 1        ?   No    ===> Cannot deliver frame

Frame 3:
            1 ≤ 3 ≤ 4     ?   Yes   ===> buffer frame 3
                                        Send ACK 3

            3 == 1        ?   No    ===> Cannot deliver frame

Same result for frame 4
```

Note:

- The **receiver window** does ***not*** **slide** forward !!!

  (because ***no*** **frame** has been ***delivered*** !!!).

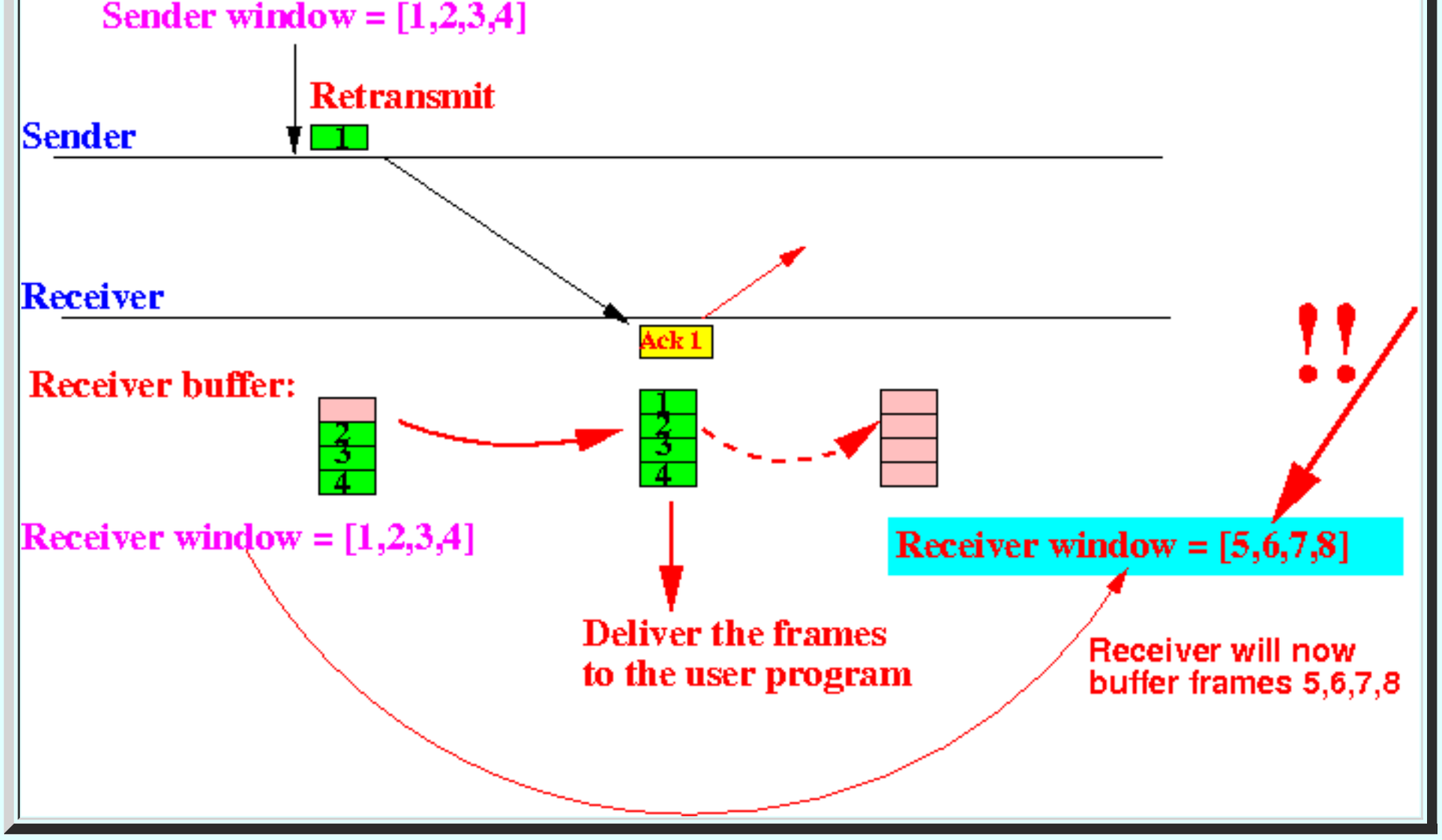
- **Example 2:** example ***continues***

  - Since the **receiver** did **not** transmit any ***ACK*** **frames** back:

    - the **sender** will **time out**

    Suppose ***after*** **the time out**, the **sender** **retransmits frame 1** and we **received** ***correctly***

    Graphically:

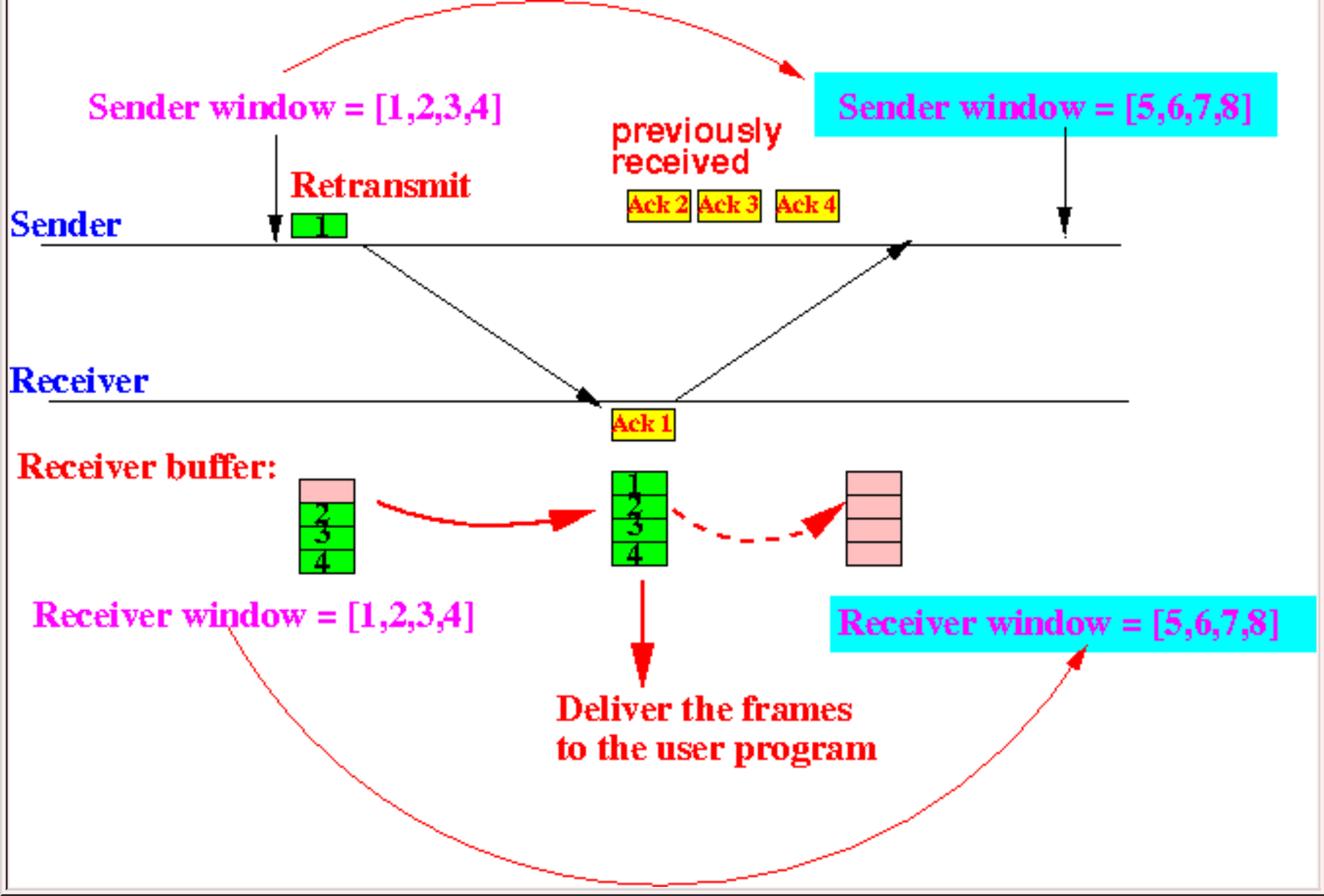


How the **receiver** process the ***received*** **frame (#1)**:

```
Receiver window = [1 .. 4]

Frame 1:
            1 ≤ 1 ≤ 4     ?   Yes   ===> buffer frame 2
                                          Send ACK 2

            1 == 1        ?   Yes   ===> deliver frames

      for ( i = 1;  i ≤ 4 ; i++ )  do:
      {
          frame i = 1 received:   Yes
                                  ===> deliver frame 1
          frame i = 2 received:   Yes
                                  ===> deliver frame 2
          frame i = 3 received:   Yes
                                  ===> deliver frame 3
          frame i = 4 received:   Yes
                                  ===> deliver frame 4
      }

 Receiver window = [5.6.7.8]
```

- **Note:**

  - When the **sender** receives the **ACK 1 frame**, the **sender window** will **move** as **follows**:



The **sender window** and **receiver window** are "***lined up***" (= **starts** with the ***same*** **seq. no.**)

- **Example Program:** (Demo above code)

Example

  - Prog file: **/home/cs455000/demo/SelAck/run**

How to run the program:

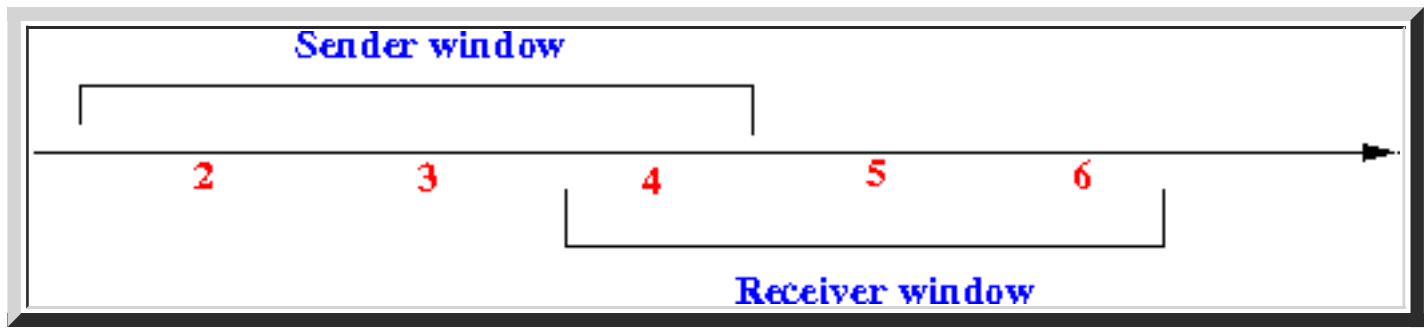
- Send **2 new frames**
- **Pause**
- Click on **frame 0** and **kill it**
- **Resume**

Observe that:

  - The **Receiver window** will **slide *only*** when the **retransmission** of **frame 1** was **received**

Sliding Windew: summary

- **Summary: "Sliding" Window**
  - A **window** is a **consecutive range of sequence number**



  - **Sender window:**

    - **Sender window** = **sequence numbers** that the **sender** can use to **label** an ***outstanding*** **frame**

      - The **sender** **must** ***buffer*** **(= retain)** an ***outstanding*** **frame** in its **send buffer**

    - **Sender Window Size (SWS)** = **# buffers** in the **sender buffer**

  - **Receiver window:**

    - **Receiver window** = **sequence numbers** of ***data*** **frames** that the ***receiver*** will **buffer** (when they are received)

    - If the ***first*** **frame** in the **receiver window** is received:

      - The **receiver** will also ***deliver*** some frames (from its **receive buffer**).
      - And **slide** the **receive window** forward by the **number of frames delivered**

    - **Receiver Window Size (RWS)** = **# buffers** in the **receiver buffer**

  - **Fact:**

    - **Sender Window Size** and **Receiver Window Size** can be **different**

      But:

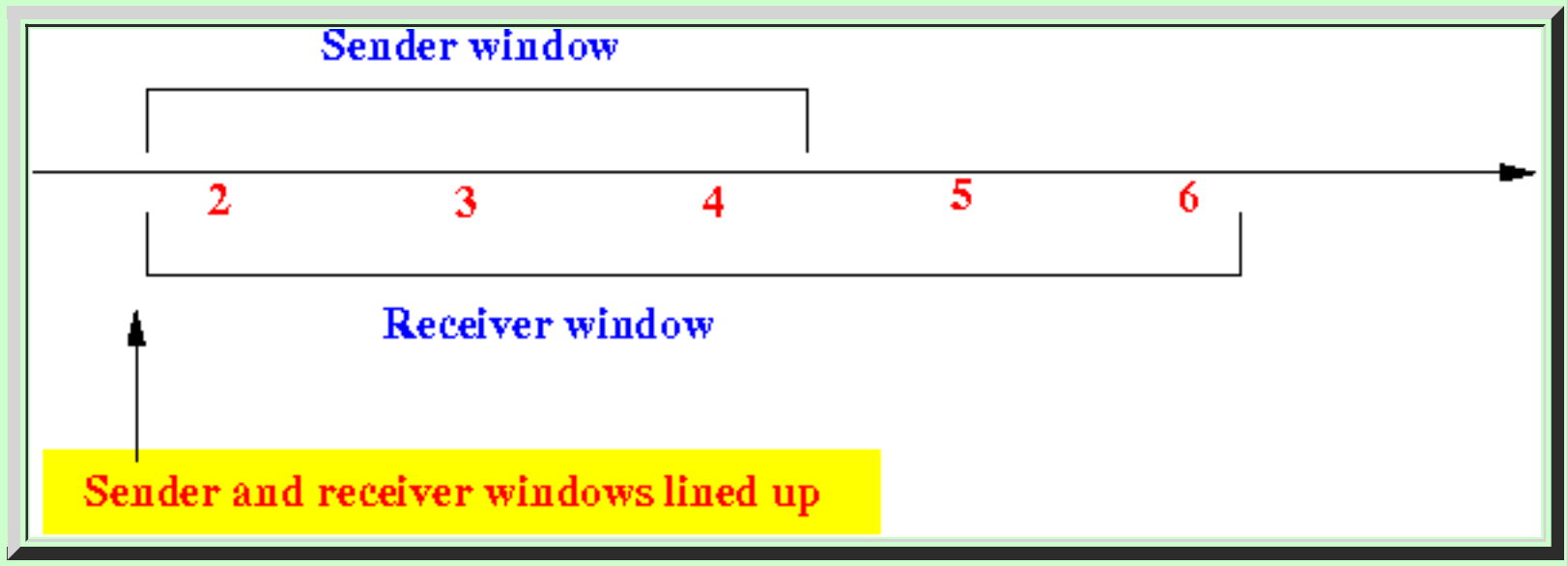
      ```
      Sender Window Size  ≤  Receiver Window Size
      ```

      because: **send buffer size ≤ receive buffer size**

      See: **click here**

- **Summary: updating the send/receive windows**
  - ***Sliding* window movement** (= **update algorithm**):

    - **Initially**, the **sender** and **receiver window *must*** have the ***same* starting number** (i.e., **"lined up"**)



    - **Sender window** will **slide (forward)** when:

      - the ***first* data frame** in the **sender window** is ***acknowledged***

        (then the **sender** can ***release*** some **buffers** !)

    - **Receiver window** will **slide (forward)** when:

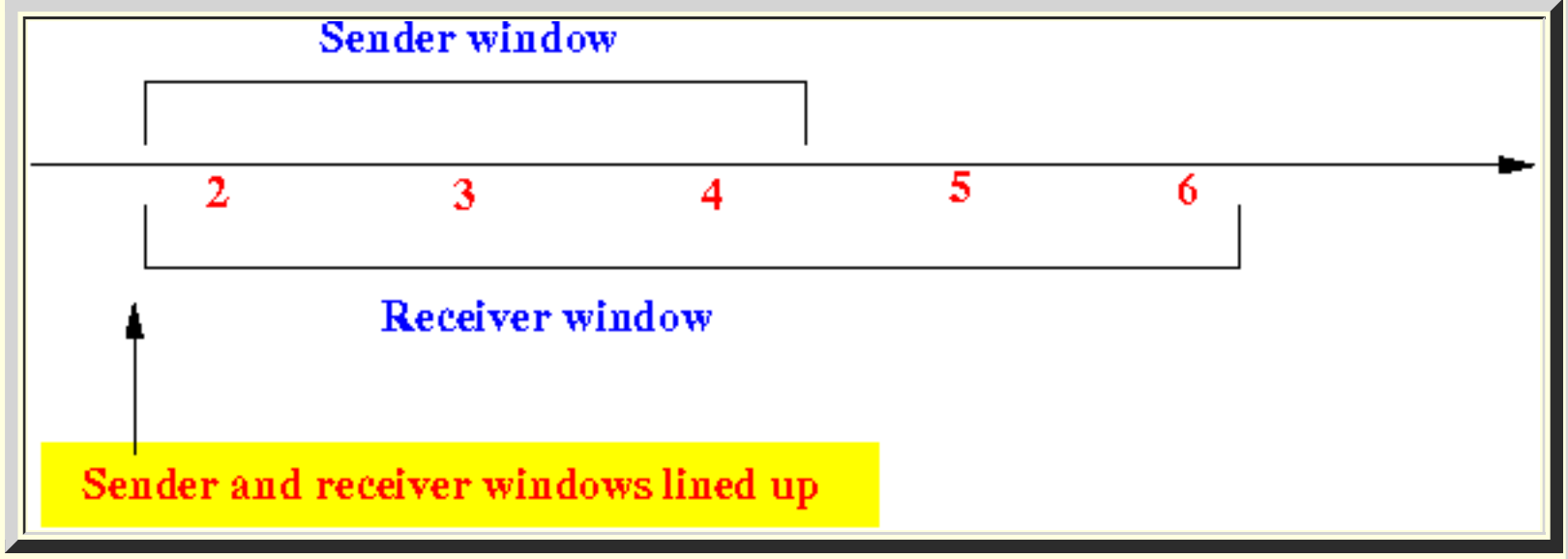
      - the ***first* data frame** in the **receiver window** is **received**

        (then the **receiver** can ***deliver*** some **data frames** !)

An important property between sender and receiver windows

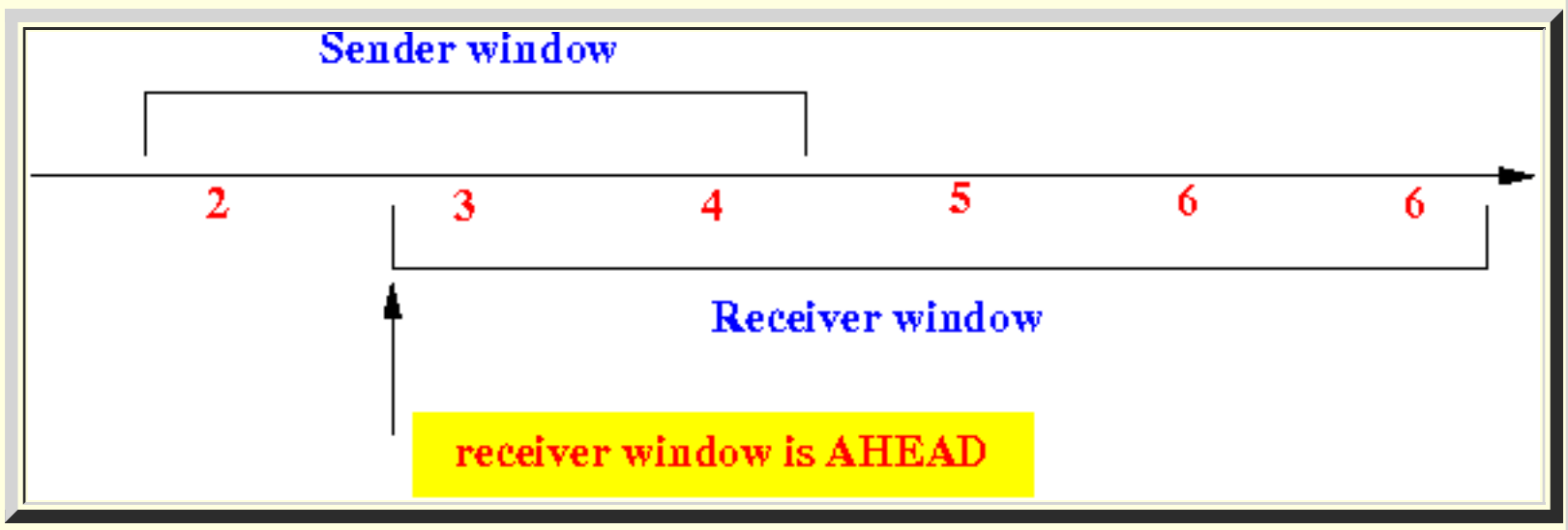
- **A *very* important property in Sliding Window protocols**
  - ***Important* property**:

    - The **receiver window** will **always** be:
      - ***lined up*** with the **sender window**:



      or:

      - ***ahead*** of the **sender window**:



I.e.:

- the **receiver window** can ***never*** fall **behind** the **sender window**

Reason:

- The **receiver window** will ***always*** **slide foward *before*** the **sender window** !!!
  - The **receiver window** will **slide** when:
    - the ***first* data frame** in the **receiver window** is **received**
  - This ***receive* event** will **trigger** the **receiver** to:
    - send the **ACK frame** for the ***first* data frame** in the ***sender* window** !!!

- The ***sender*** **window** will ***only*** **slide** when:

  - this **ACK frame** is **received**

  The **sender window** can ***only*** **slide** ***later*** in **time** !!!!

- **Example Program:** (Demo above code)

  Example

  - Prog file: **/home/cs455000/demo/SelAck/run**

  How to run the program:

  - Send **1 new frames**

Acknowledgement schemes

- **Acknowledgement schemes used in reliable communication protocols**
  - There are **2 kinds** of **acknowledgement schemes** used in **reliability protocols**:

1. ***Selective* acknowledgement**:

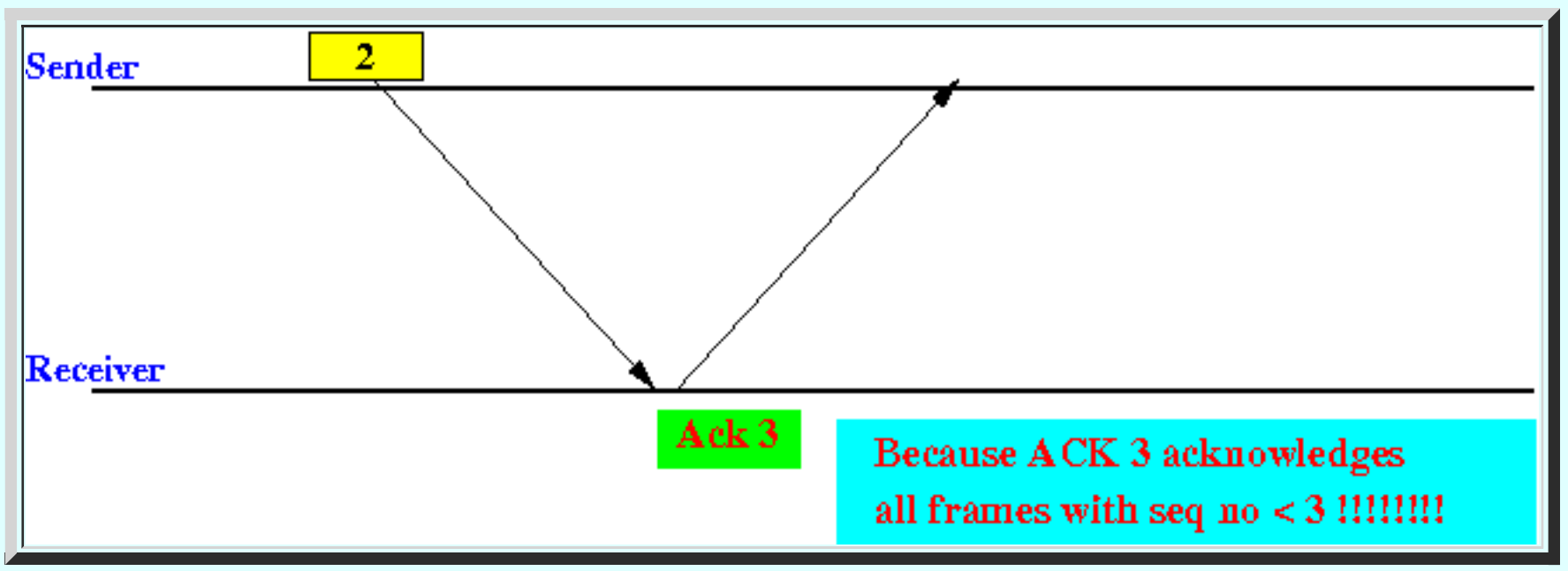
   - **ACK *n*** (***n*** = ***receive* sequence number**) acknowledges:
     - the **(correct) reception** of the **data frame** with **send sequence number = *n***

2. ***Cumulative* acknowledgement**:

   - **ACK n** acknowledges:
     - the **(correct) reception** of ***all* data frames** with **send sequence number < n**

       I.e.: the **(correct) reception** of ***all* data frames** with **send sequence number ≤ (*n*–1)**

Example:



Note:

- **ACK 3** means:
  - The **first** of the ***new* frames** for the **receiver** has **sequence number #3**

- **Strength/weakness of the cumulative ACK scheme**
  - **Fact:**
    - **ACK *n*** will ***also* acknowledge**

- **All data frames** that have been **acked** by **ACK (*n*–1)**

- **ACK (*n*–1)** will ***also*** **acknowledge**

  - **All data frames** that have been **acked** by **ACK (*n*–2)**

- **And so on**....

In other words:

- Tranmitting **ACK *n*** will ***implicitly***:

  - ***re-transmitting*** **all** the **ACK** frames **ACK (*n*–1)**, **ACK (*n*–2)**, ...

- **Strength** of the ***cumulative*** **ACK scheme**:

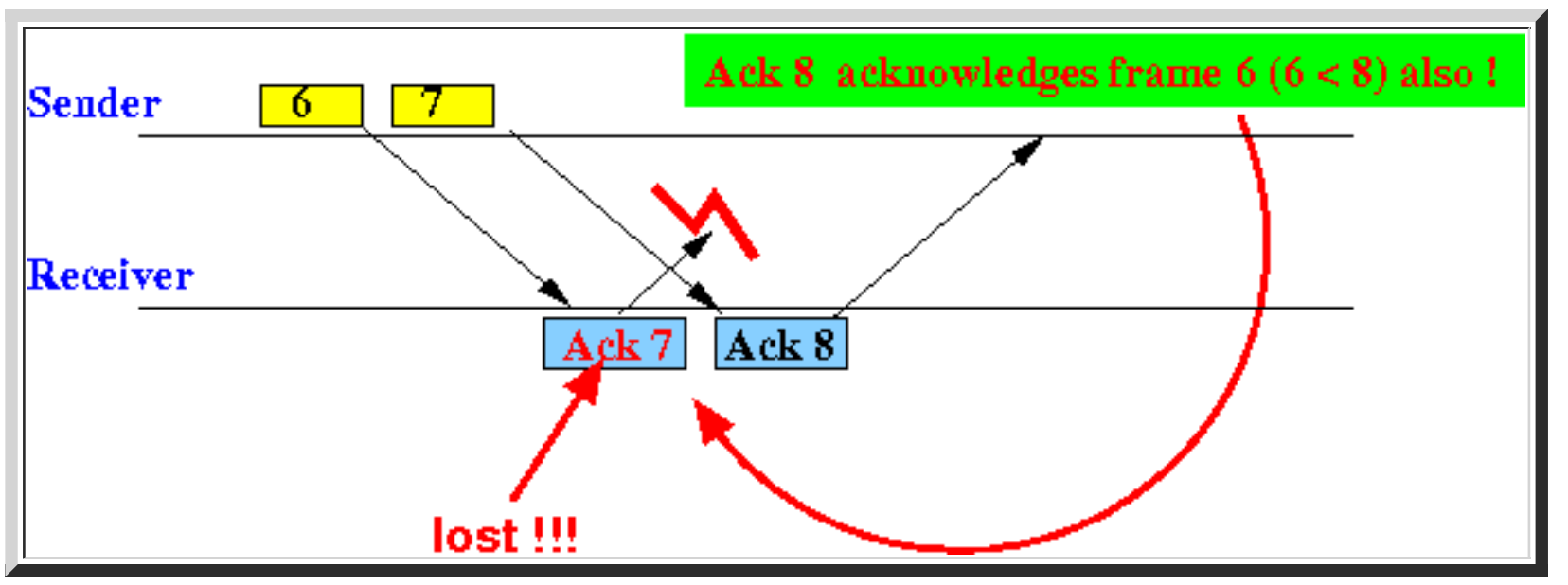
  - The **ACK frames** in the ***cumulative*** **ACK scheme** are ***very*** **well protected**

In other words:

- If the **ACK *n* frame** was ***lost***, then:

  - the **lost ACK *n* frame** can be **recovered** by the **ACK (*n*+1) frame** !!!

Example: if **ACK 7** is **lost**, then ***data*** **frame 6** is ***also*** **acknowledged** if **ACK 8** is **received**



- **Weakness** of the ***cumulative*** **ACK scheme**:

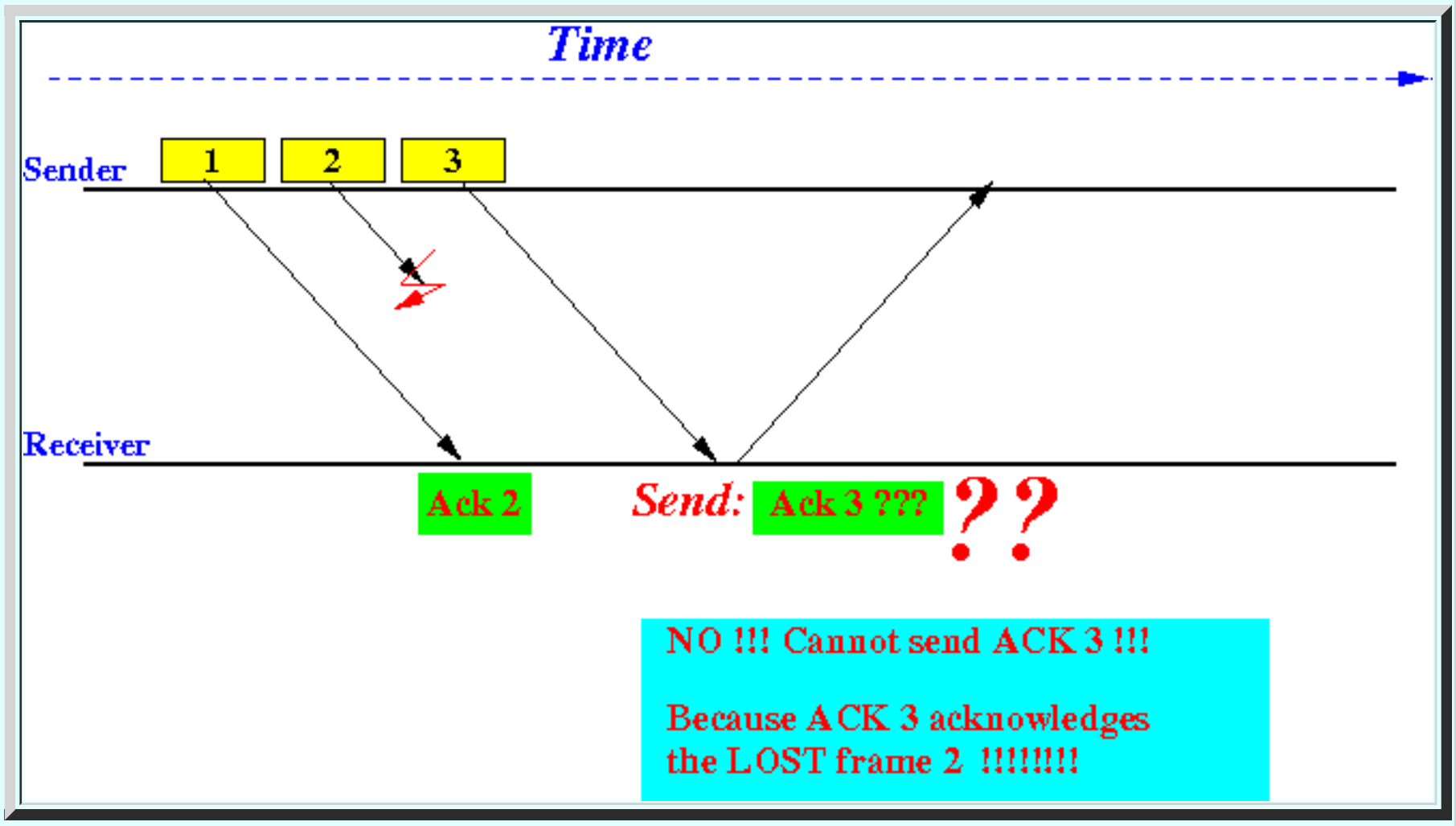
  - The **receiver** can ***not*** **acknowledge**:

    - ***correctly*** **data frames** that are received ***out-of-order***

Example:

- The **sender** transmits **frames 1,2,3**

  Frame 1 was **lost**
  Frame 2, 3 were received ***correctly***



Then:

- The **receiver** **can *not*** send **ACK 3** !!!

  (because **ACK 3** will ***also*** acknowledge the ***lost*** **frame 1** !!!)

- **Consequence:**

  - The **sender** may **time out** on the ***correctly*** **received** frames (**2 and 3**)

  - This ***can*** lead to ***inefficiency*** because **sender** will ***retransmit*** some **data frames** ***unneccassarily***

- **Strength/weakness of the Selective ACK scheme**

  - **Strength:**

    - **Extremely *efficient***:

      - The **sender** will ***only*** **retransmit** the **unacknowledged (corrupted) frames**

Weakness:

- The **ACK frames** are ***not*** **protected** from **loss**

  - An **ACK** can be **lost (= corrupted)**

- Lost ACK frames will cause (unnecessary) *retransmission* and reduce efficiency)

- **Which method is more popular ?**
  - **As of 2014:**

    - The *cumulative acknowledgement* method is **preferred** for long distance communication **because:**

      - The error rate is relatively *high*

        So you **want** to protect the acknowledgement messages !!!

  Fact:

    - The Transport Control Protocol (TCP) of the Internet Protocol (IP) uses *cumulative* acknowledgement

  - **Note:**

    - With the advent of high speed and low error rate *fiber optic* cables, the *selective* acknowledgement method is being considered to replace the **cumulative ack** scheme.

      (But it's very difficult to switch protocol on a *global* scale...

      I don't expect the Internet to use *selective* ACK before I'm dead...)

How the *Sliding Window* protocol will operate when using *cumulative* acknowledgement

- **Review: the necessity for the *receiver* to transmit (old) ACK frames**
  - In the **dicsussion** of **Stop-and-Wait**, we **learned** that:

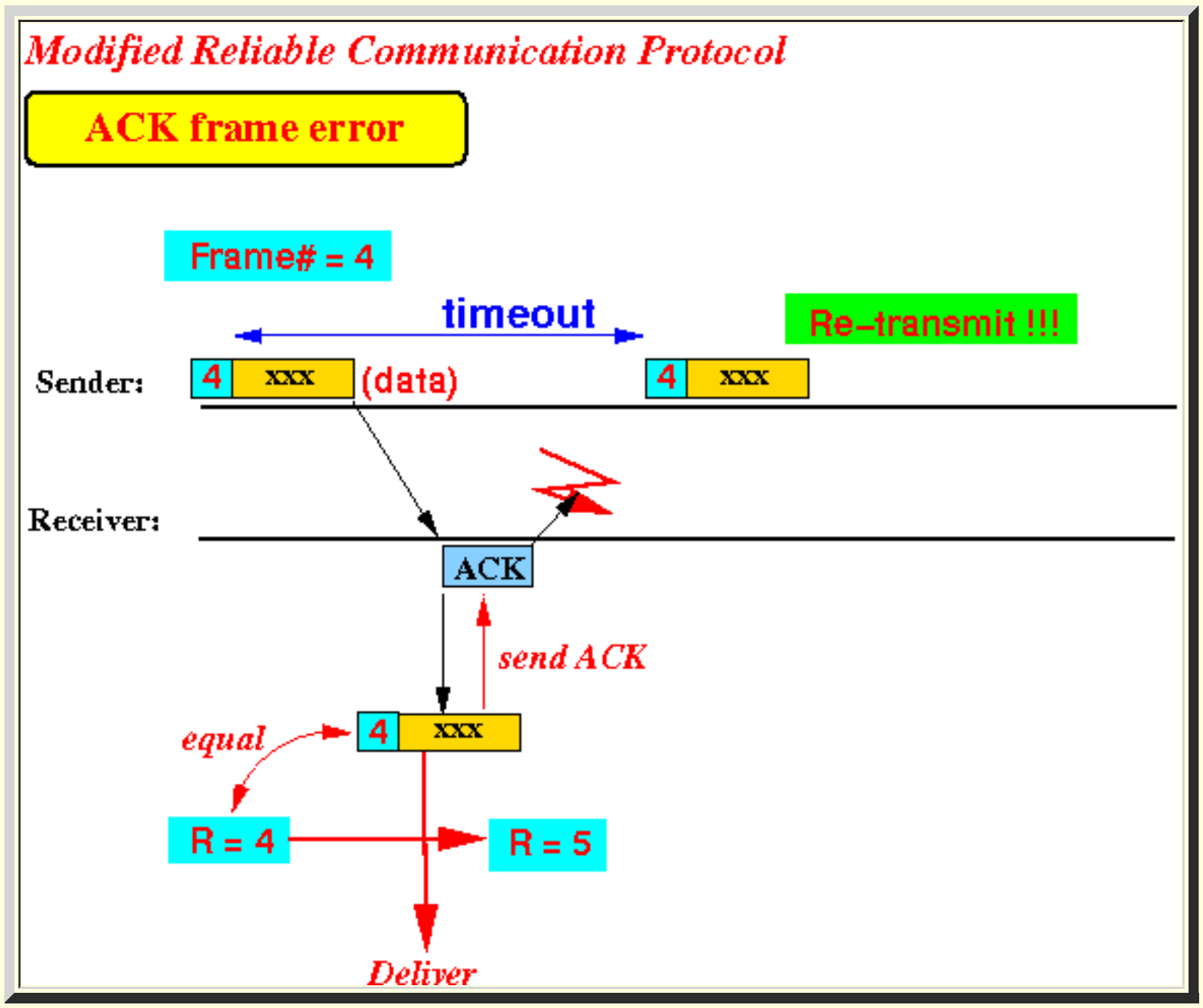
    - When the **receiver** receives a ***re-transmission*** (of an ***old*** **data frame**:
      - The **receiver** has transmit a **"old" ACK frame**

      This **must** be done to:

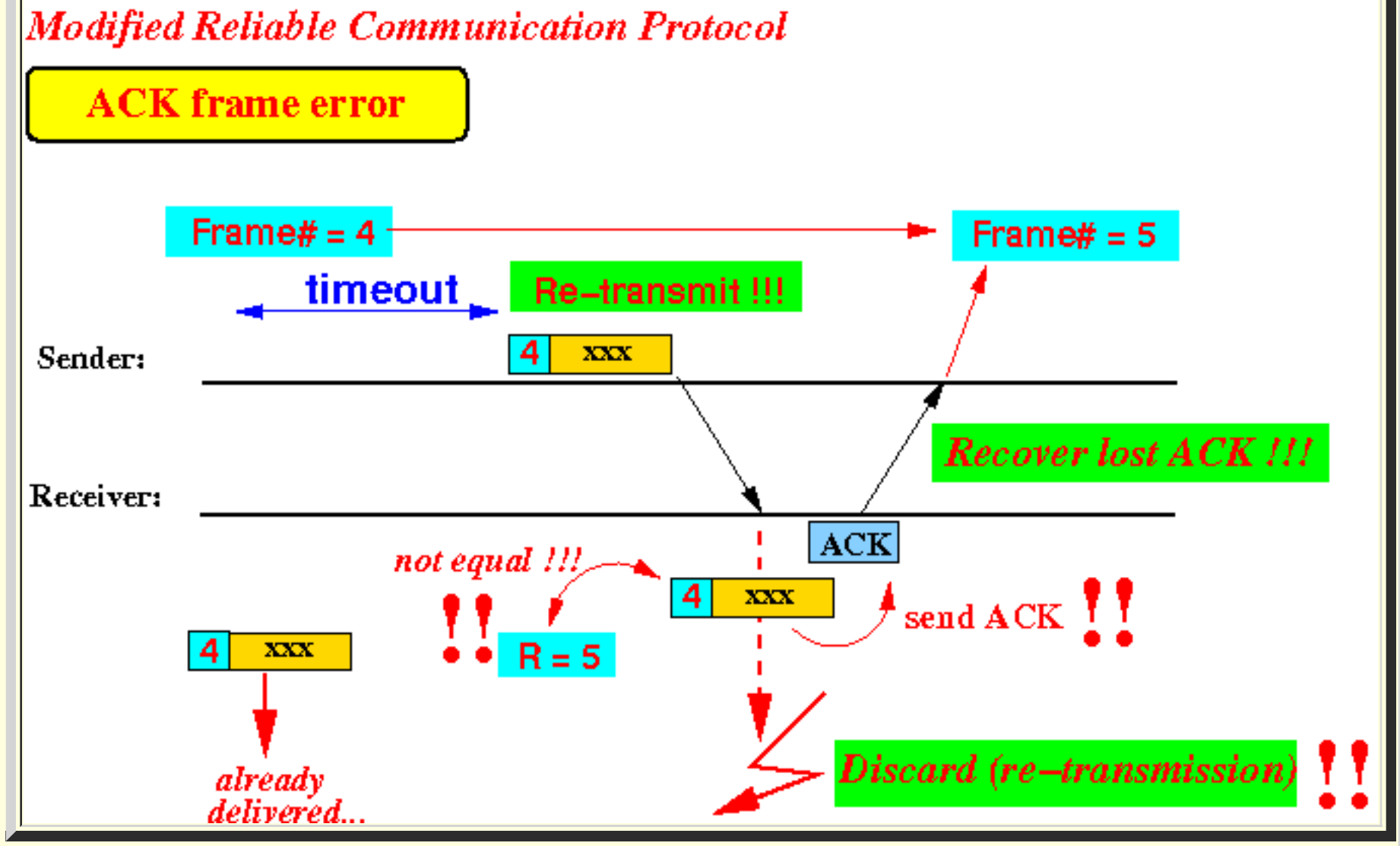
      - **Recover** from the ***lost*** of an ***ACK*** **frame**

    Example:

    - The **ACK frame** is **lost**:



- The **sender** will ***time out*** and **re-transmit** the **data frame**:

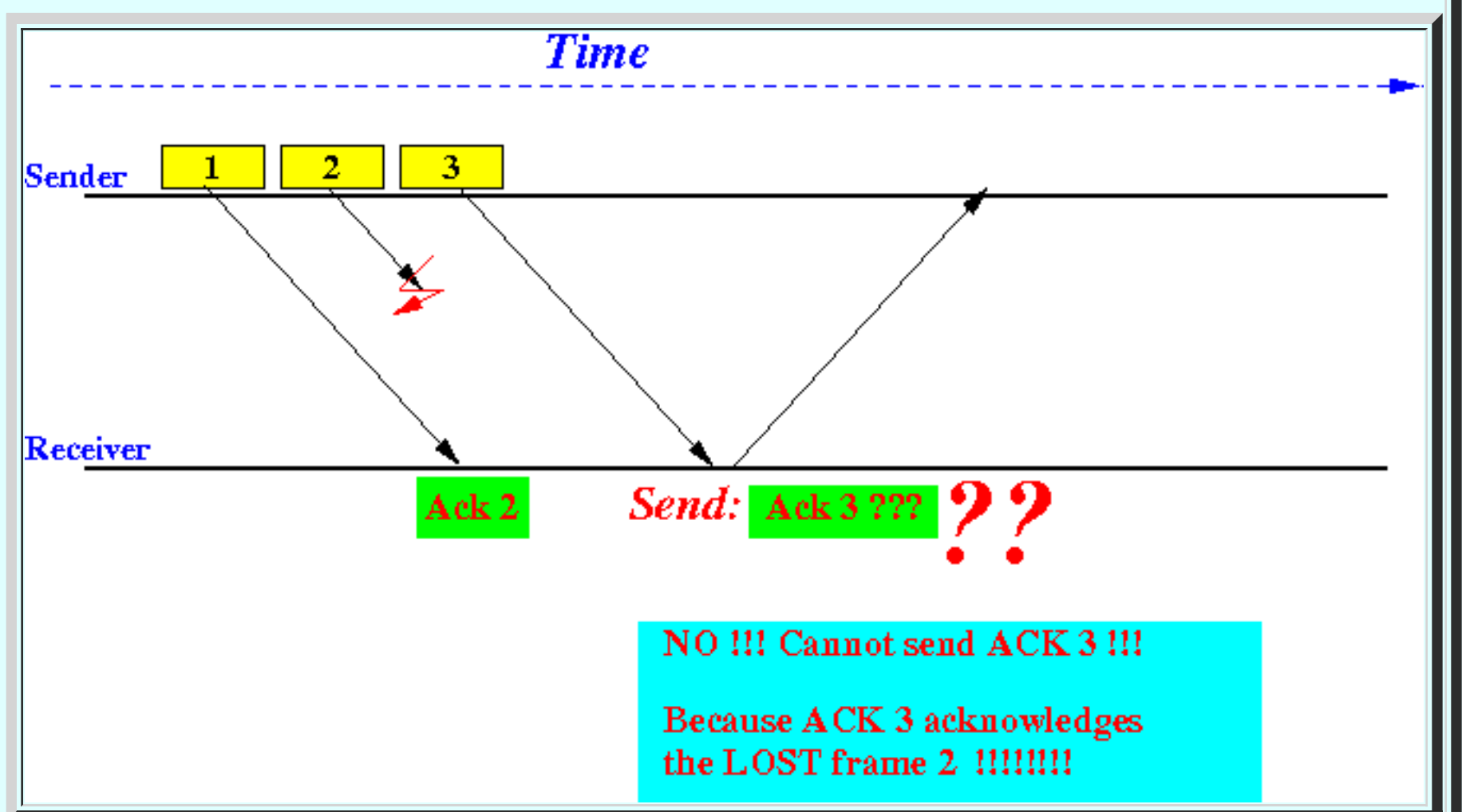


The **receiver** ***must*** transmit an ***old*** **ACK** to **receover** from the **ACK frame lost** !!!!

- **Cumulative ACK scheme: be careful what you acknowledge !!!**
  - **Recall** that in the ***cumulative*** **ACK scheme**:
    - **ACK *n*** will **acknowledge *old* data frames *also*** !!!

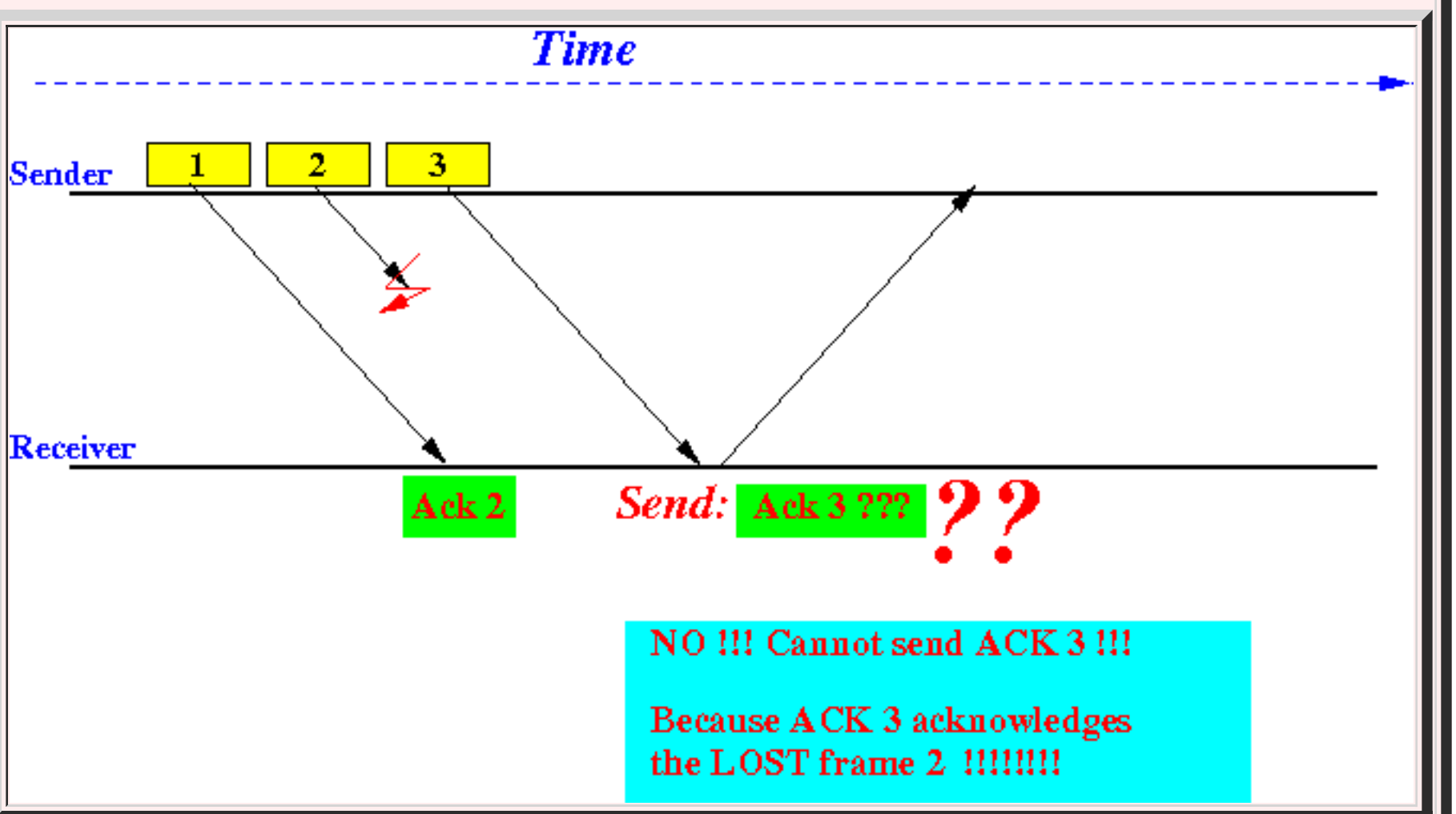
Important example:

- **ACK 3** will ***also*** **acknowledge** the ***lost*** **data frame #2**:



- **Conclusion:**

  - When the receiver receives an ***out-of-order*** **data frame**:

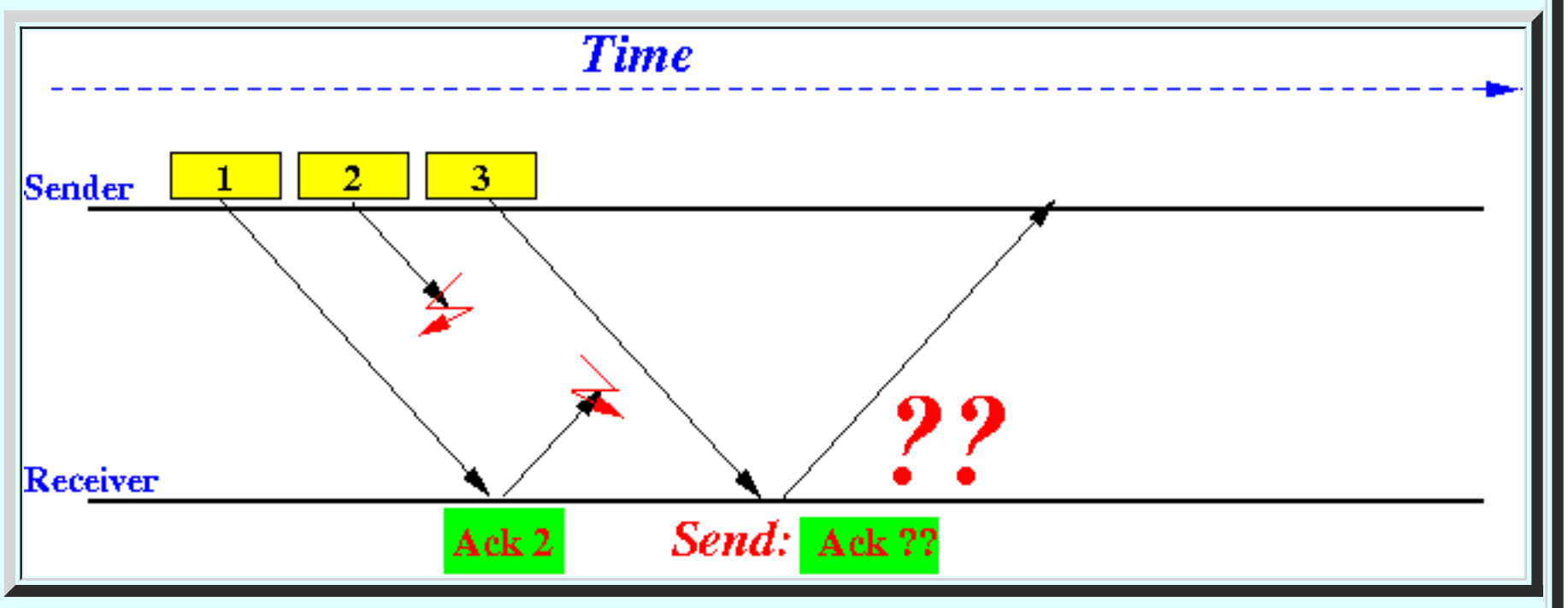


    the **receiver** must **transmit** a **ACK frame** that does **not** acknowledges a ***missing*** **data frame**

- **How to use a *cumulative* ACK frame in case of error**

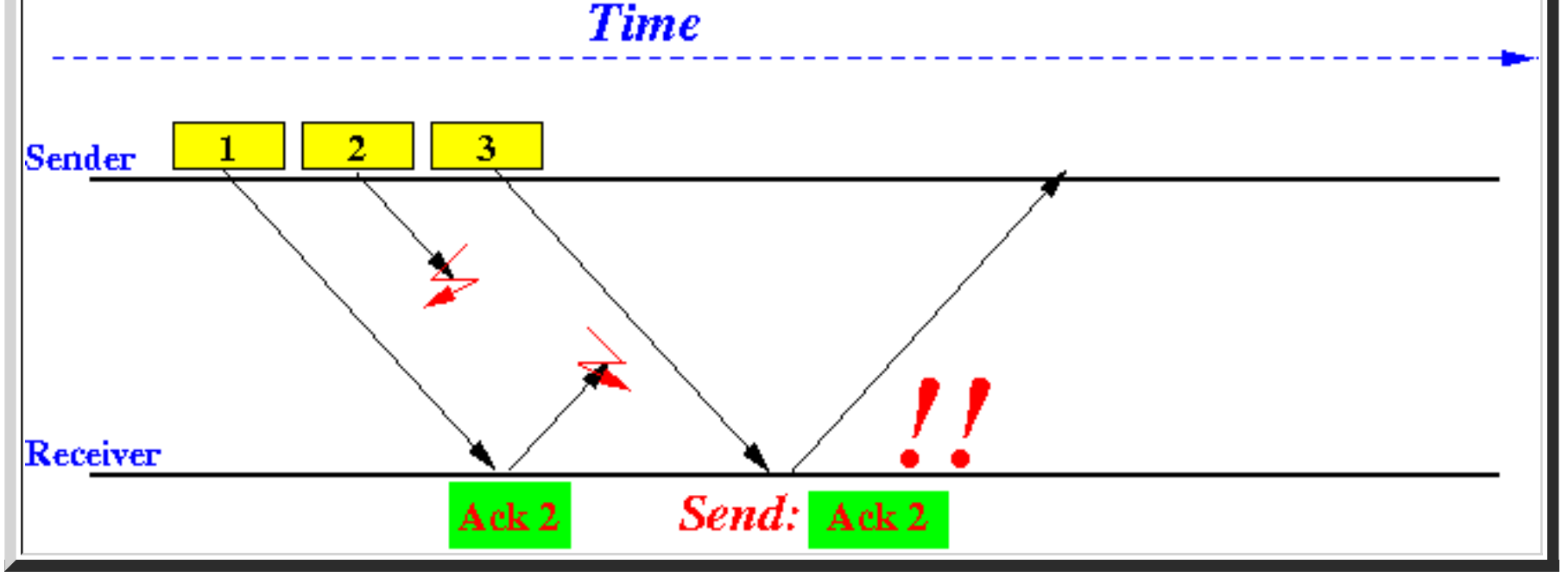
  - **Problem statement:**

    - **What ACK sequence number** does the **receiver** transmit in the **following situation**:



  - **Solution:** ***repeat*** the ***last*** **ACK**

    Example:



Repeating the **last** **ACK 2** serves **2 purposes**:

- **ACK 2 can** be used to ***recover*** a ***possible lost*** **ACK** frame !!!

- **ACK 2** does ***not*** **acknowledge** the ***lost (missing)*** **frame 1** !!!

  (**because** **ACK 2** acknowledges **data frames** with **send seq no ≤ 1**)

The *Sender* Window Algorithm using cumulative ACKs

- **Recall: Sender window**
  - **Sender window:**

    - **Sender window** = the **serie** of ***send*** **sequence numbers** that the **sender** can use to **identify** a **outstanding frame**

  - **Fact:**

    - The **sender** must **retain** all **outstanding data frames** in its **send buffer**

      (In case the **receiver** requests the **data frame** be ***re-transmitted***)

- **How the sender window changes (= "moves")**
  - **Update algorithm** for the **sender window:**

```
Let Sender Window = [ A .. B ]

Sender receives an ACK frame

Let X = recv seq no in received ACK frame

---

if (  X  >  A  )
{
   /* ====================================================
      Frames A, A+1, ... , X-1 have been acknowledged !!
      ==================================================== */
   for ( each Y = A, A+1, ..., X-1 ) do
   {
       /* ----------------------------------
          Frame Y is acknowledged !!!
          ---------------------------------- */
       Release buffer for frame Y
    }

    Sender Window = [X .. X + (B-A) +1)]
                    // I.e.: set sender window to [X .....]
}
```

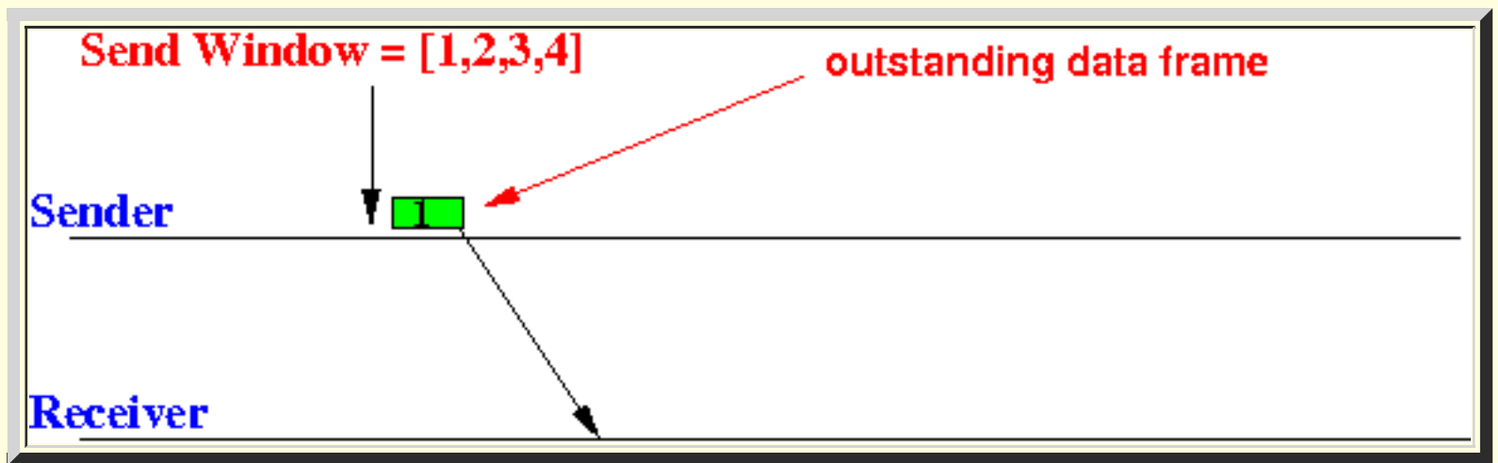
(Use black board to show examples in class !!!)

  - **Example 1:**

    - Suppose the **current** ***sender*** **window** is:

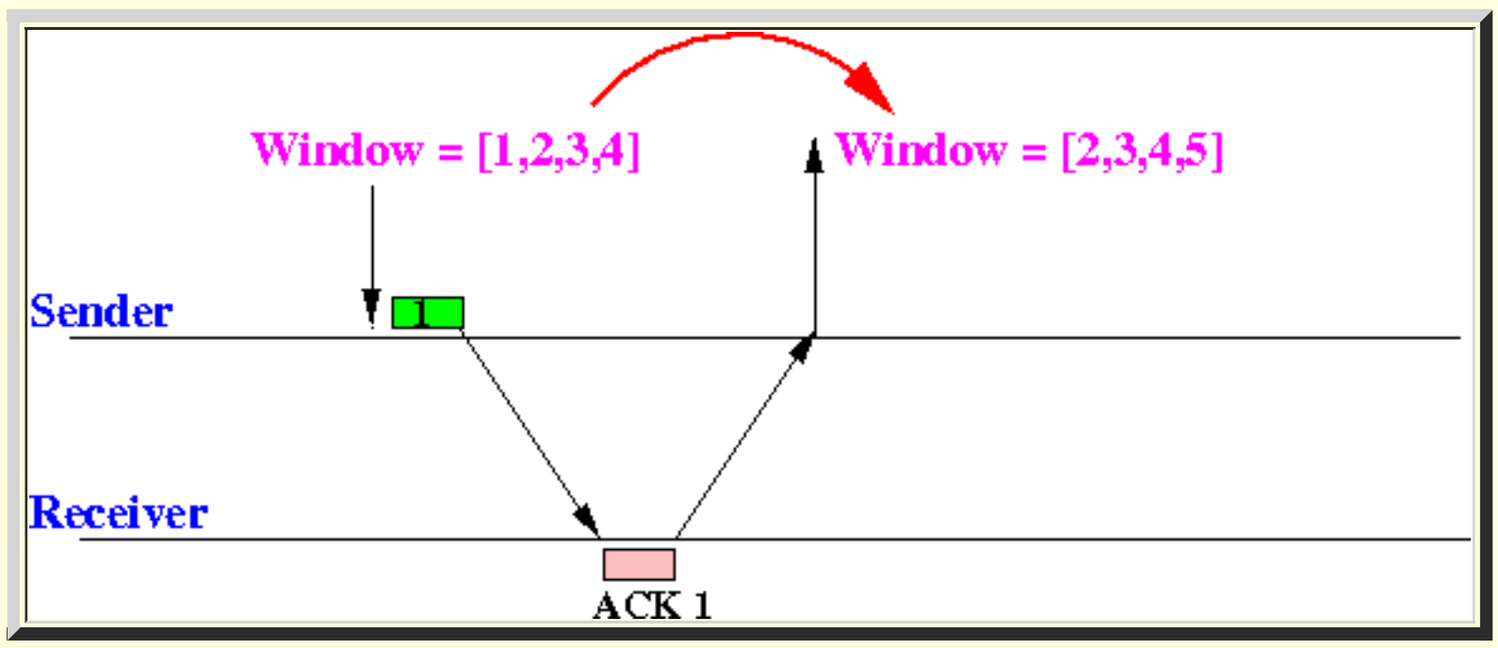
```
Sender window = [1, 2, 3, 4]
```

    - The **sender** transmits the **next data frame**:



The ***next*** **data frame** will use the ***first*** available **number** in the **sender window** is its **send seq. no.**!!!

- When the **sender** **receives the** **ACK frame**, **the** **send window** **will be** **updated** **to** **[2,3,4,5]:**



- **Example 2:**

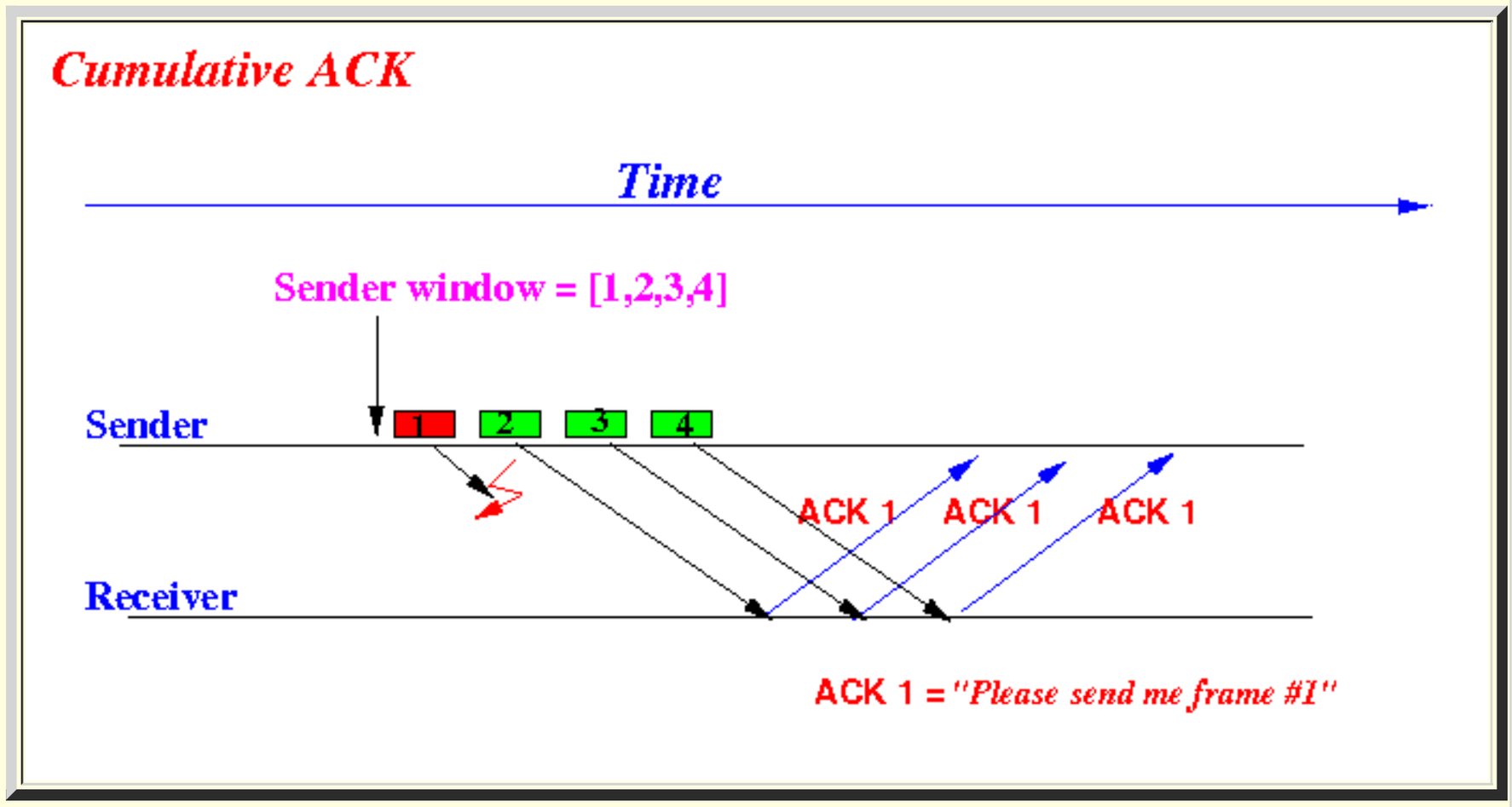
  - **Suppose:**

    - **Sender window** = **[1,2,3,4]**

    Events:

    - The **sender** transmits **frames** **1,2,3,4**
    - **But**: **Frame 1** is **lost**

    - The **receiver** transmits back **ACK frames 1,1,1 !!!**



because:

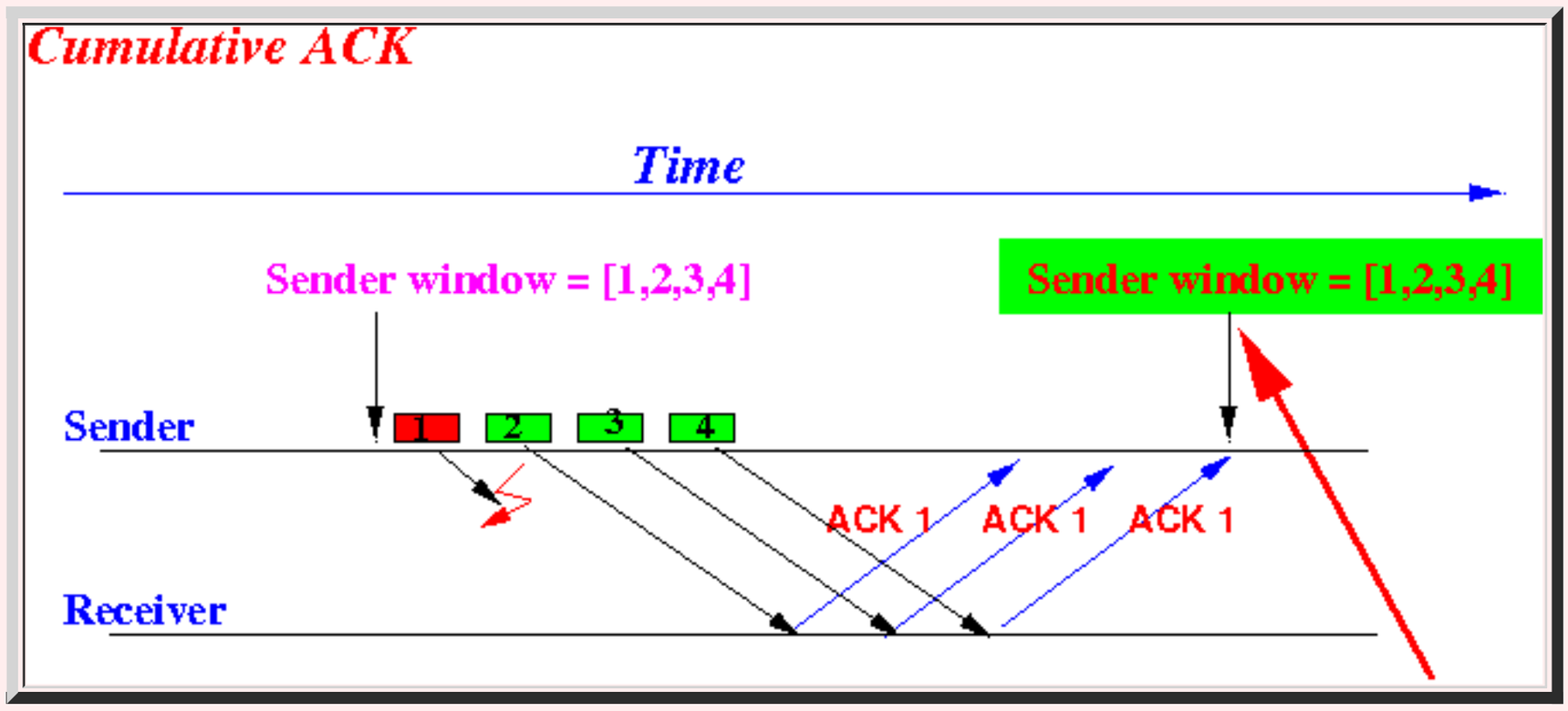
- **ACK 1** does ***not*** acknowledge the ***lost*** **frame 1**

How the **sender** process the ***received*** **ACK frames**:

```
Sender window = [1 .. 4]

ACK 1:     1 > 1  ?   No (do not update sender window)

ACK 1:     1 > 1  ?   No (do not update sender window)
```

```
ACK 1:     1 > 1   ?    No (do not update sender window)
```

Graphically:



Note:

- The **sender window** does ***not*** **slide** forward !!!

  (because ***no*** **send buffer** has been ***released*** !!!).

- **Example 3:** **previous example *continues***

- The **sender** **times out** and **retransmits** **frame 1** and **frame 1** was **received** ***correctly***

- **Current sender/receiver windows:**

  - **Sender window** = **[1,2,3,4]**

  Events:

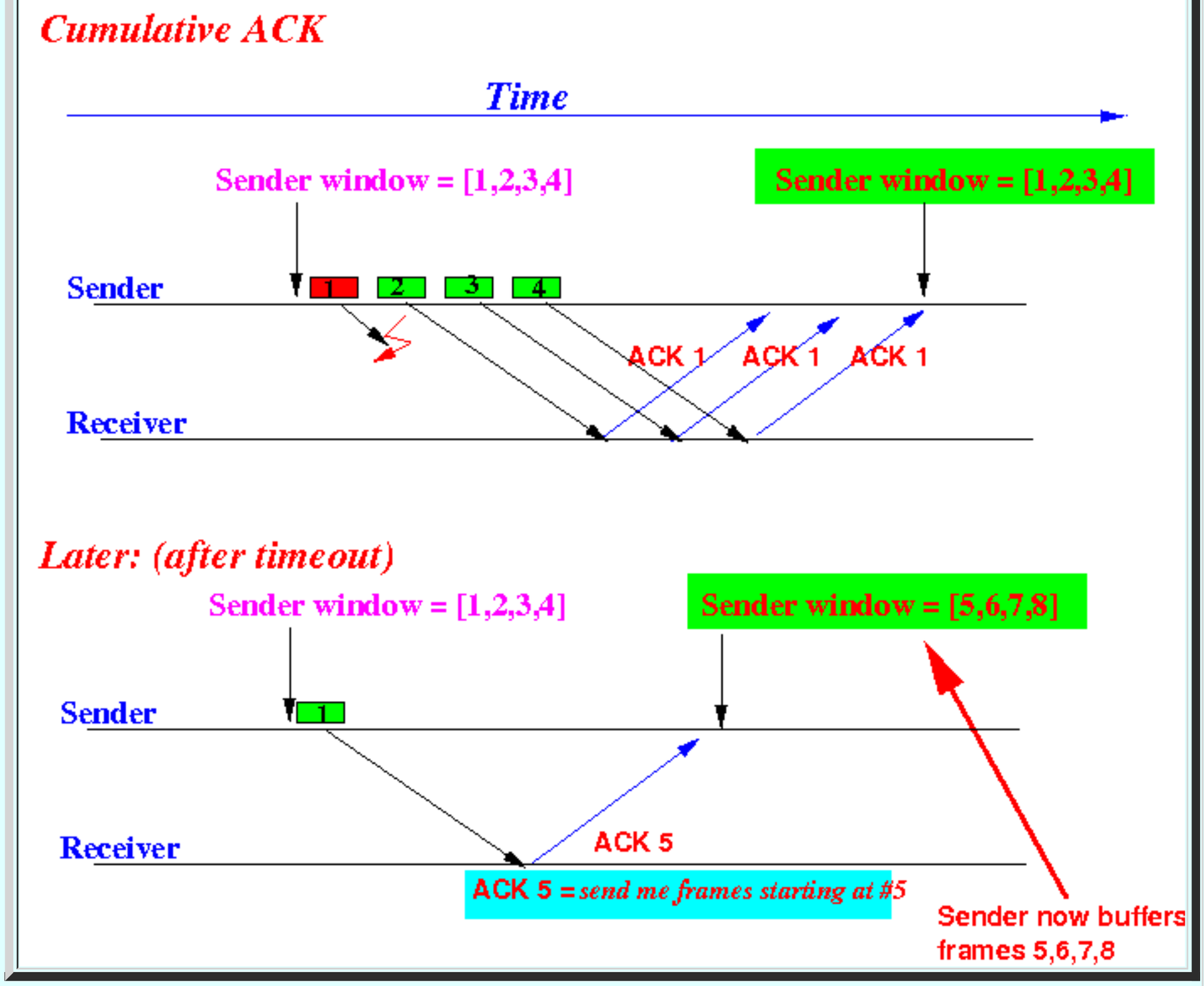
  - The **sender** retransmits **frame 1**
  - **Frame 1** is **received**

  - The **receiver** transmits **ACK frame 5 !!!**

How the **sender** process the ***received*** **ACK frame (#5)**:

```
Send window = [1 .. 3]

ACK 5:     5 > 1 ?    Yes

       for each frame Y = 1, 2, 3, 4 do:

           Y = 1  ===> Release buffer for frame 1
           Y = 2  ===> Release buffer for frame 1
           Y = 3  ===> Release buffer for frame 1
           Y = 4  ===> Release buffer for frame 1

       Sender Window = [5 .. 8]

(Because ACK 5 acknowledges frames 1,2,3,4 !!!)
```

Graphically:



Result:

- The **sender** will ***release*** **all 4 buffered frames (1,2,3,4)**
  - **Now** the **sender window** will **slide** forward by **4 positions**

  The **sender** will **now** buffer the **data frames 5,6,7,8**

The *Receiver* Windew Algorithm using cumulative ACKs

- **Review: Receiver window**
  - **Recall** that:
    - **Receiver buffer** = **buffer** in the **receiver** to store ***undeliverable*** **data frames**
  - **Receiver window:**
    - **Receiver window** = the **sequence numbers** of the **data frames** that the **receiver** will **store (save)** in the **receiver buffer**

- **How the receiver window changes (= "moves") with cumulative ACKs**
  - **Update algorithm** for the **receiver window**:

```
Let Receiver Window = [ A .. B ]

Receiver receives a data frame;

Let X = send sequence number in received data frame;
```

```
if ( X == A )
{
   Save the received data frame in the receiver buffer;

    /* ---------------------------------------
       Deliver buffered data frames !!!
       --------------------------------------- */
    i = A;

    while ( data frame i is received )
    {
       Deliver frame i
       i++;
    }

    /* ===============================================
       Now: i = seq no of first missing data frame
       =============================================== */
    Receiver Window = [i .. i+(B-A)+1]
                           // Advance receiver window !!!
    Send ACK i;
}
else if ( A  <  X  ≤  B )
{
   Save the received data frame in the receiver buffer;

   Send ACK A;     // Frame #A is still missing
}
else
{
   Discard frame !!!     // This must have been an old frame

   Send  ACK A;          // Recover possibly lost ACK frames
}
```

  - **Example 1:**
    - **Suppose:**
      - **Sender window** = [1,2,3,4]
      - **Receiver window** = [1,2,3,4]

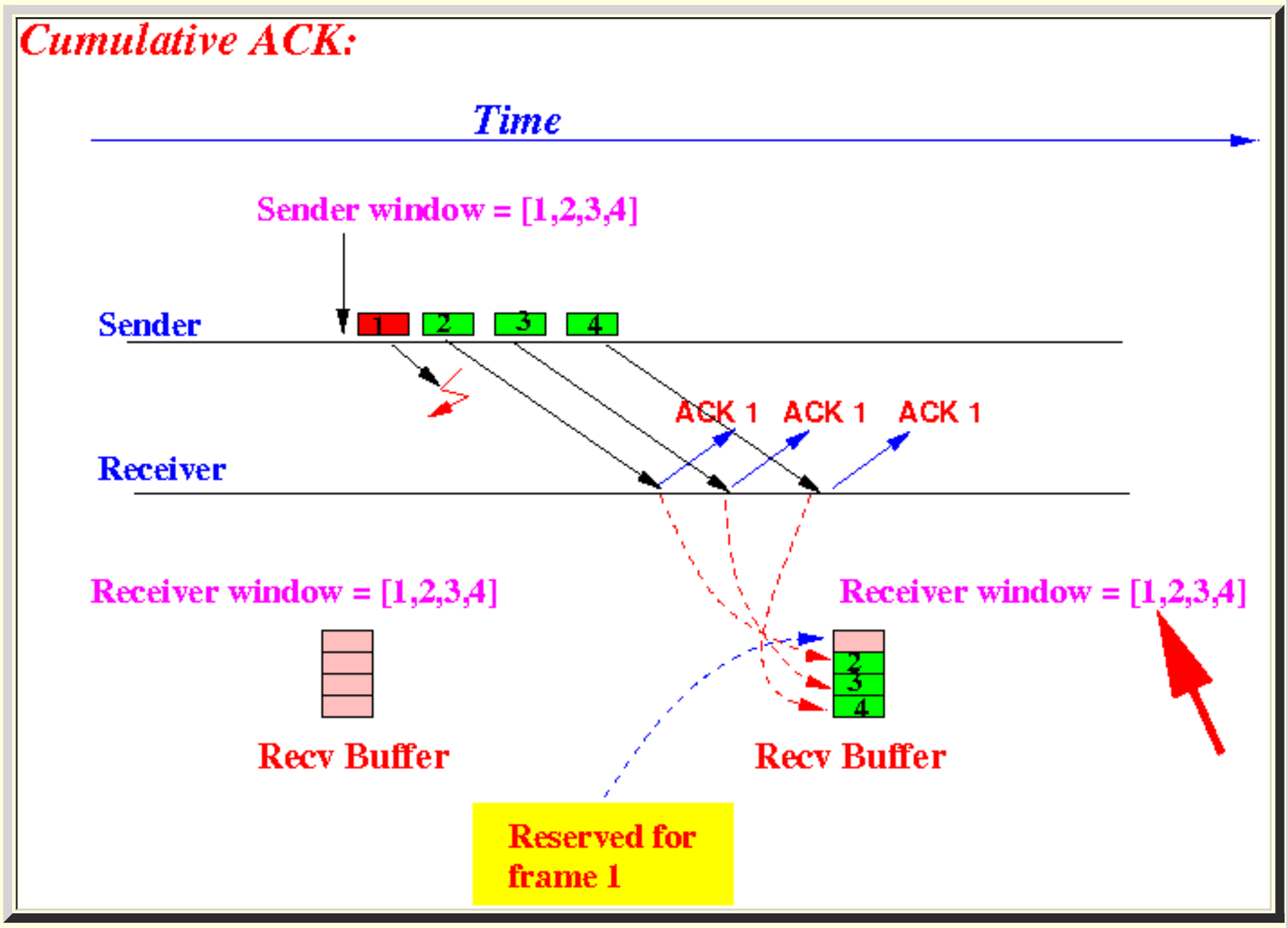
      Events:

- The **sender** transmits **frames 1,2,3,4**
- **But**: **Frame 1** is **lost**

How the **receiver** process the ***received*** **frames**:

```
Receiver window = [1 .. 4]

Frame 1:    lost

Frame 2:    2 == 1          ?   No
            1 < 2 ≤ 4       ?   Yes   ===> buffer frame 2
                                      Send ACK 1

Frame 3:    3 == 1          ?   No
            1 < 3 ≤ 4       ?   Yes   ===> buffer frame 3
                                      Send ACK 1

Frame 4:    4 == 1          ?   No
            1 < 4 ≤ 4       ?   Yes   ===> buffer frame 4
                                      Send ACK 1
```

Graphically:



Note:

- The **receiver window** does ***not*** **slide** forward !!!

  (because ***no*** **frame** has been ***delivered*** !!!).

- **Example 2:** **example *continues***

  - Since the **receiver** did **not** send ***ACK*** for **frame #1**:

    - the **sender** will **time out**

    Suppose ***after*** **the time out**, the **sender** **retransmits** **frame 1** and it is **received** ***correctly***

  - **Current sender/receiver windows:**

- **Sender window** = [1,2,3,4]
- **Receiver window** = [1,2,3,4]

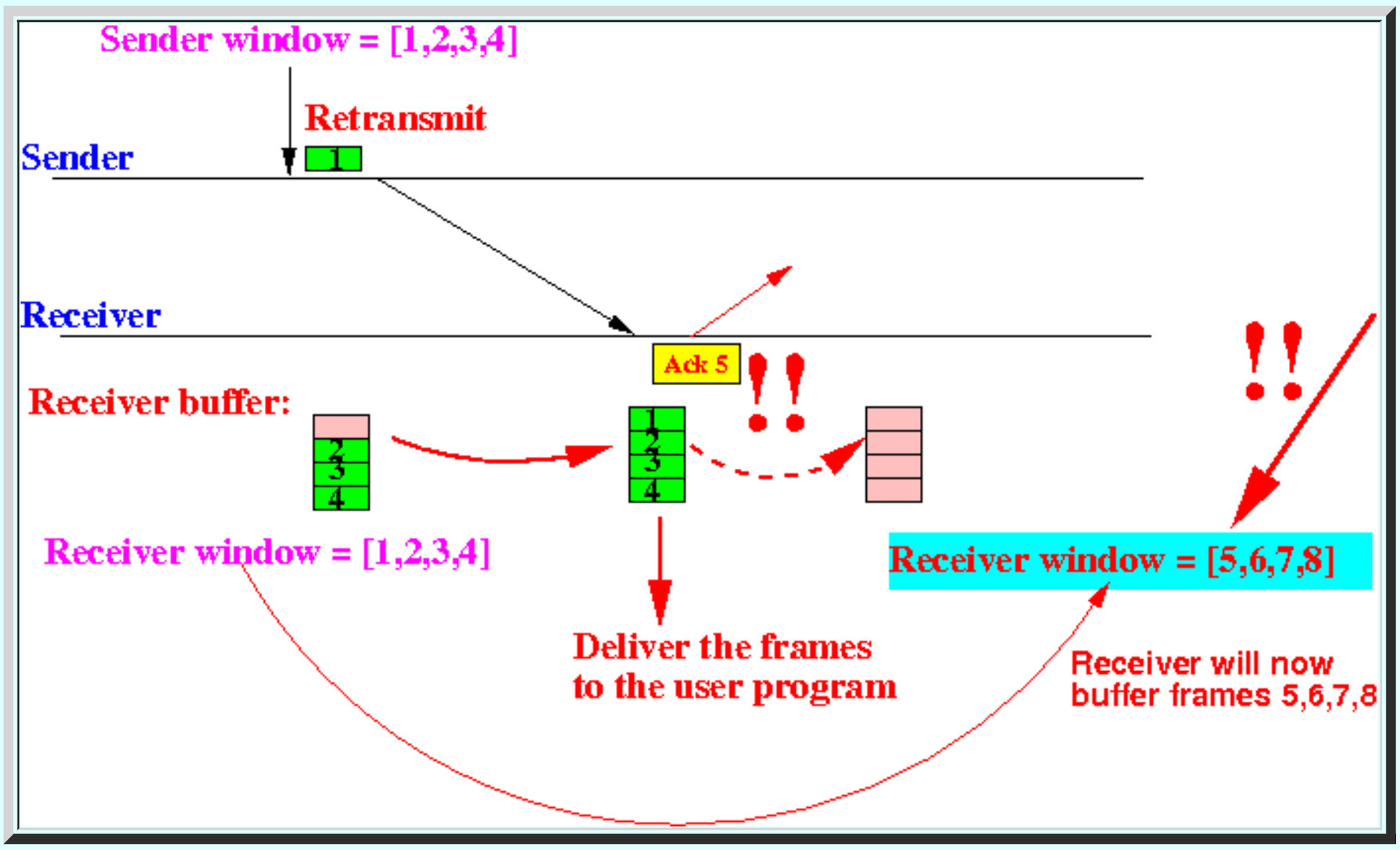
Events:

- The **sender** retransmits **frame 1**
- **Frame 1** is **received** ***correctly***

How the **receiver** process the ***received*** **frame (#1)**:

```
Receiver window = [1 .. 4]

Frame 1:    1 == 1        ?   Yes !!

      ===> Buffer frame 1
      ===> Send ACK 1

      i = 1;

      while ( frame i received ) loop:

          i = 1:     is received ?  Yes
                                    ===> deliver frame 1
                                    ===> i++ (i=2)

          i = 2:     is received ?  Yes
                                    ===> deliver frame 2
                                    ===> i++ (i=3)

          i = 3:     is received ?  Yes
                                    ===> deliver frame 3
                                    ===> i++ (i=4)

          i = 4:     is received ?  Yes
                                    ===> deliver frame 4
                                    ===> i++ (i=4)

      Receiver Window = [5 .. 8]

      Send  ACK 5 !!!
```

Graphically:



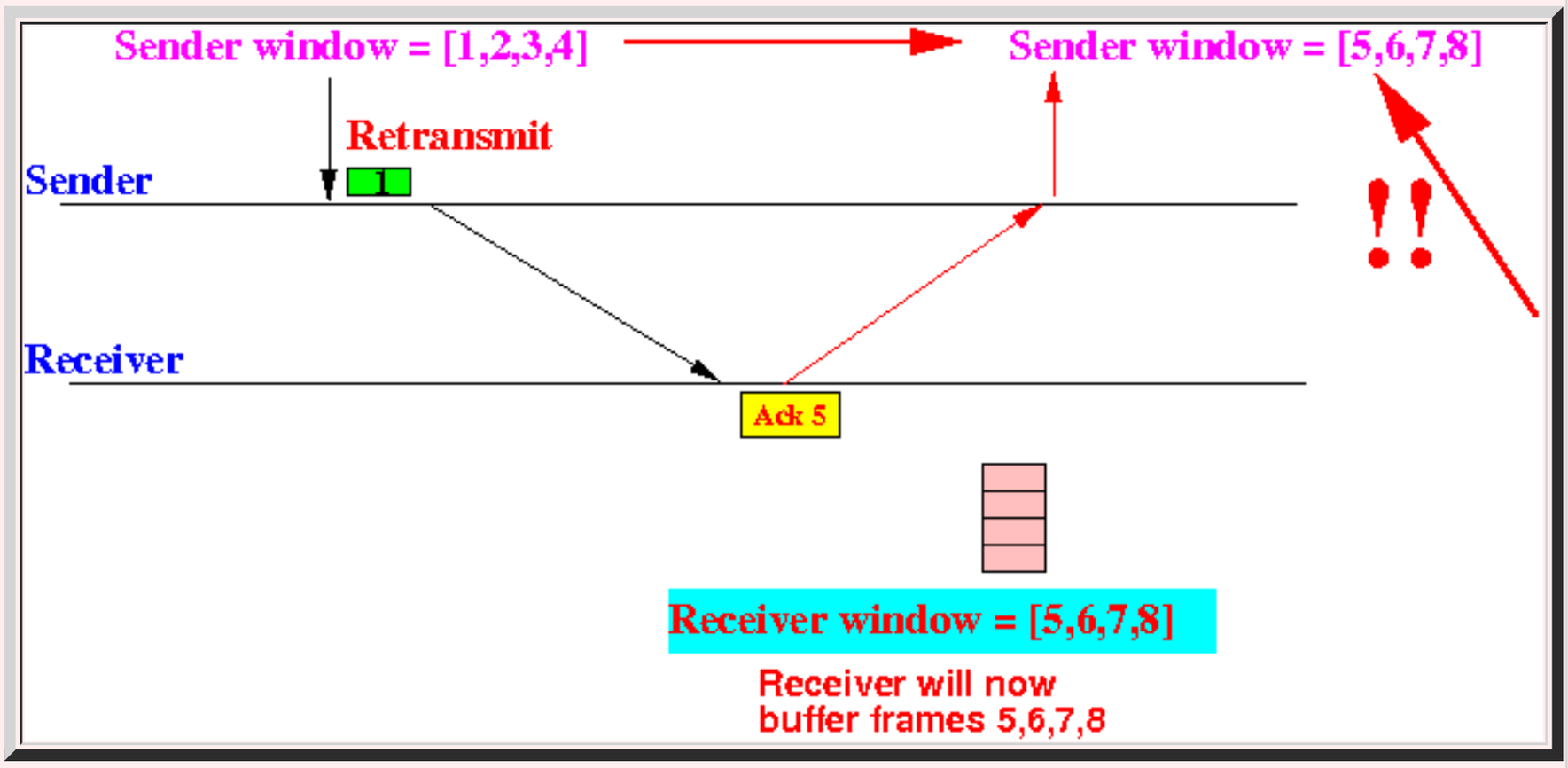
Result:

- The **receiver** will ***deliver*** **frames 1,2,3,4** to the **user program**
  - **Now** the **receiver window** will **slide** to **[5,6,7,8]**

The **receiver** will **now** **buffer** the **data frames 5,6,7,8** !!!

- **Note:**

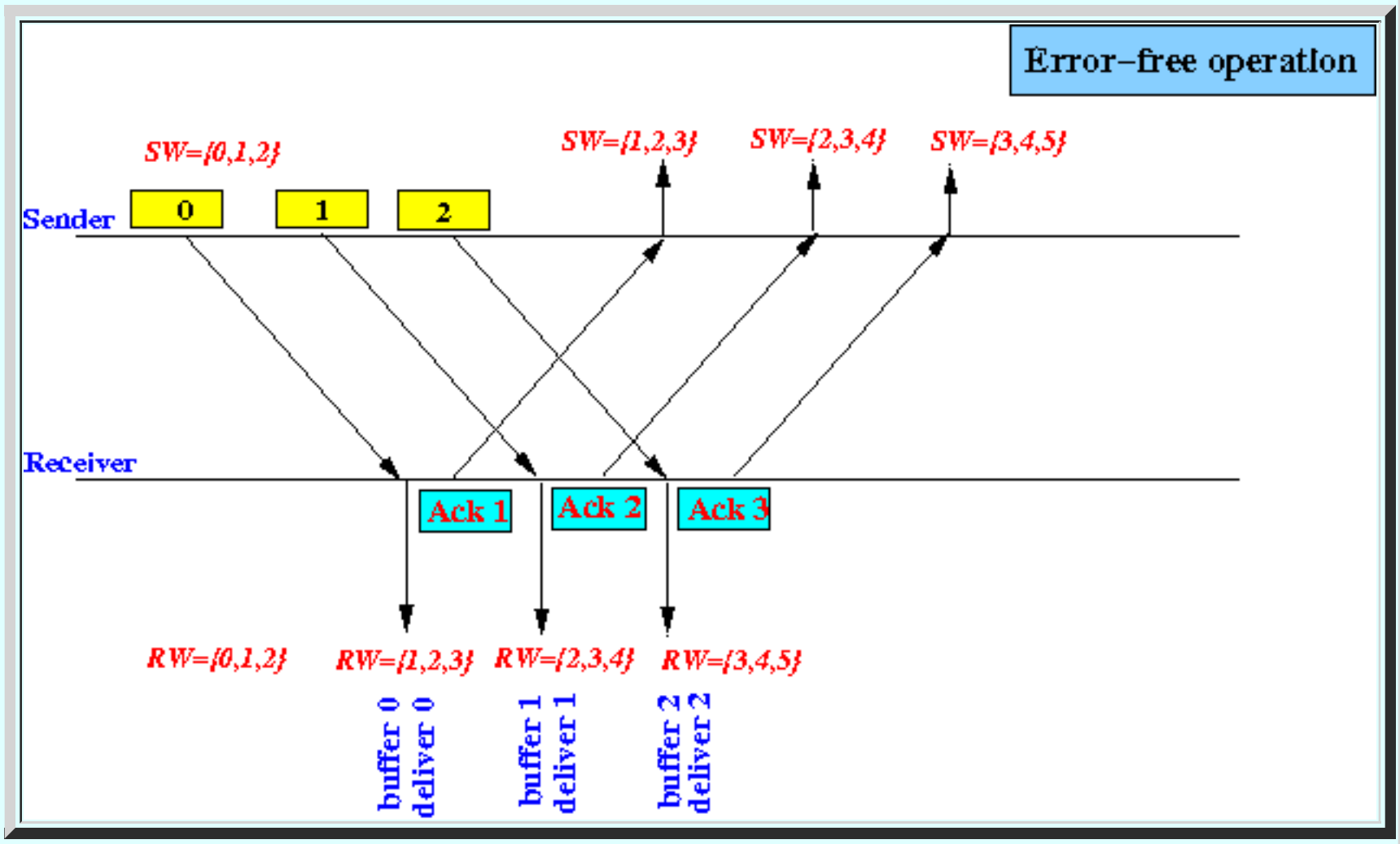
  - When the **sender** receives the **ACK 5 frame**, the **sender window** will **move** as **follows**:



The **sender window** and **receiver window** are "***lined up***" (= **starts** with the ***same*** **seq. no.**)

Sliding Window with cumulative ACK: Error-free operation

- ***Sliding Window* with Cumulative ACK Example: Error Free Operation**
  - **Error-free operation:**
    - **Sender:**
      - Send window = **[0, 1, 2]**
    - **Receiver:**
      - Receive window = **[0, 1, 2]**
    - **Example:**



Comments:

- Watch for this **progression**:
  1. **Sender** sends frame ***x***
  2. **Receiver** receives frame ***x***
     - **Receiver *delivers*** the frame ***x***
     - ***Receiver* window** slides **forward**
     - **Receiver** sends **ACK *x+1***
  3. **Some times *later***, the **sender** will receive **ACK *x+1***:

- ***Sender* window *now*** slides **forward**

- **Notice** that:

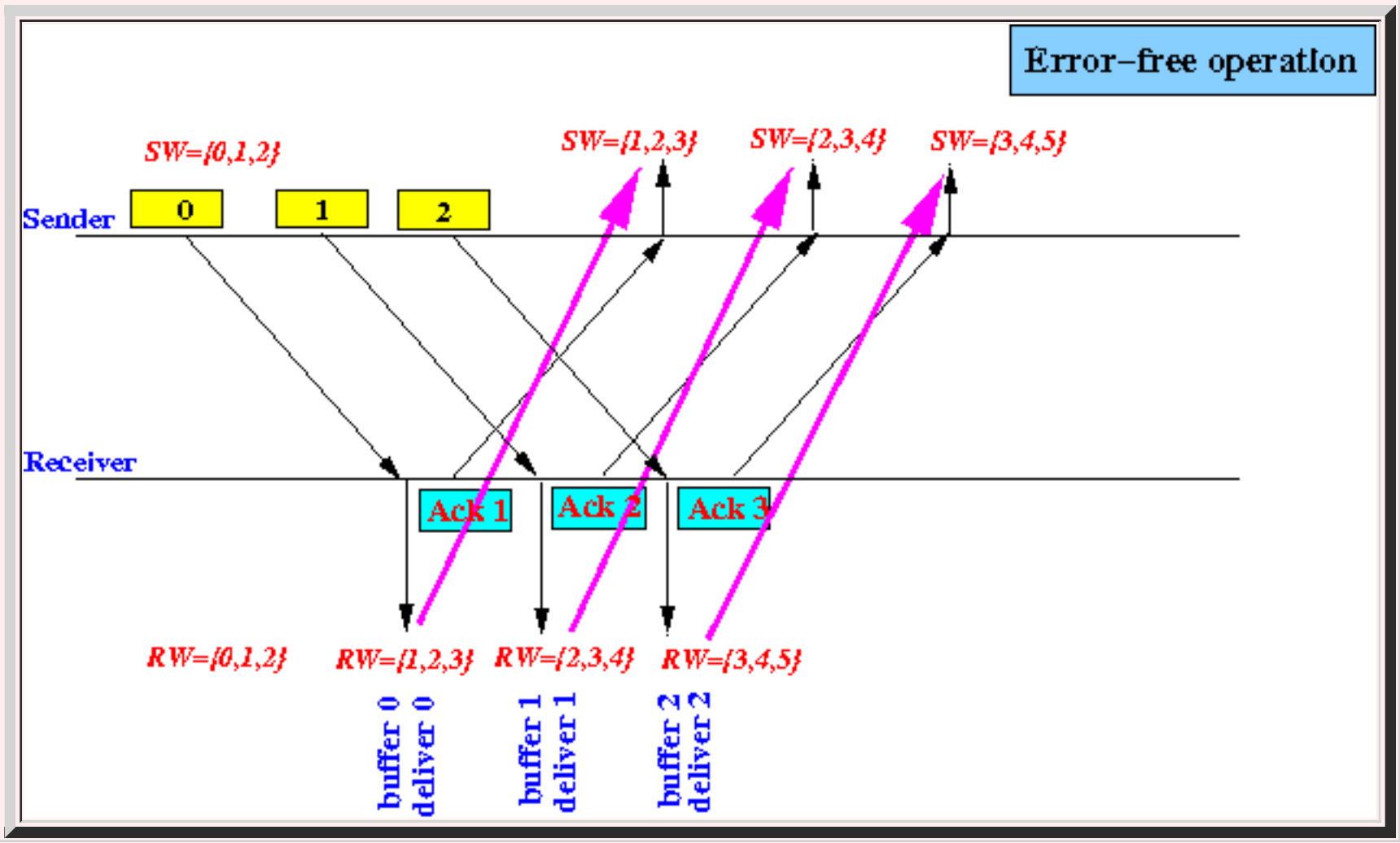
  - At ***all* times**:

    - The ***receiver* window** is

      - is ***aligned* with** ***or***
      - ***ahead* of**

      the ***sender* window**

  Observe:



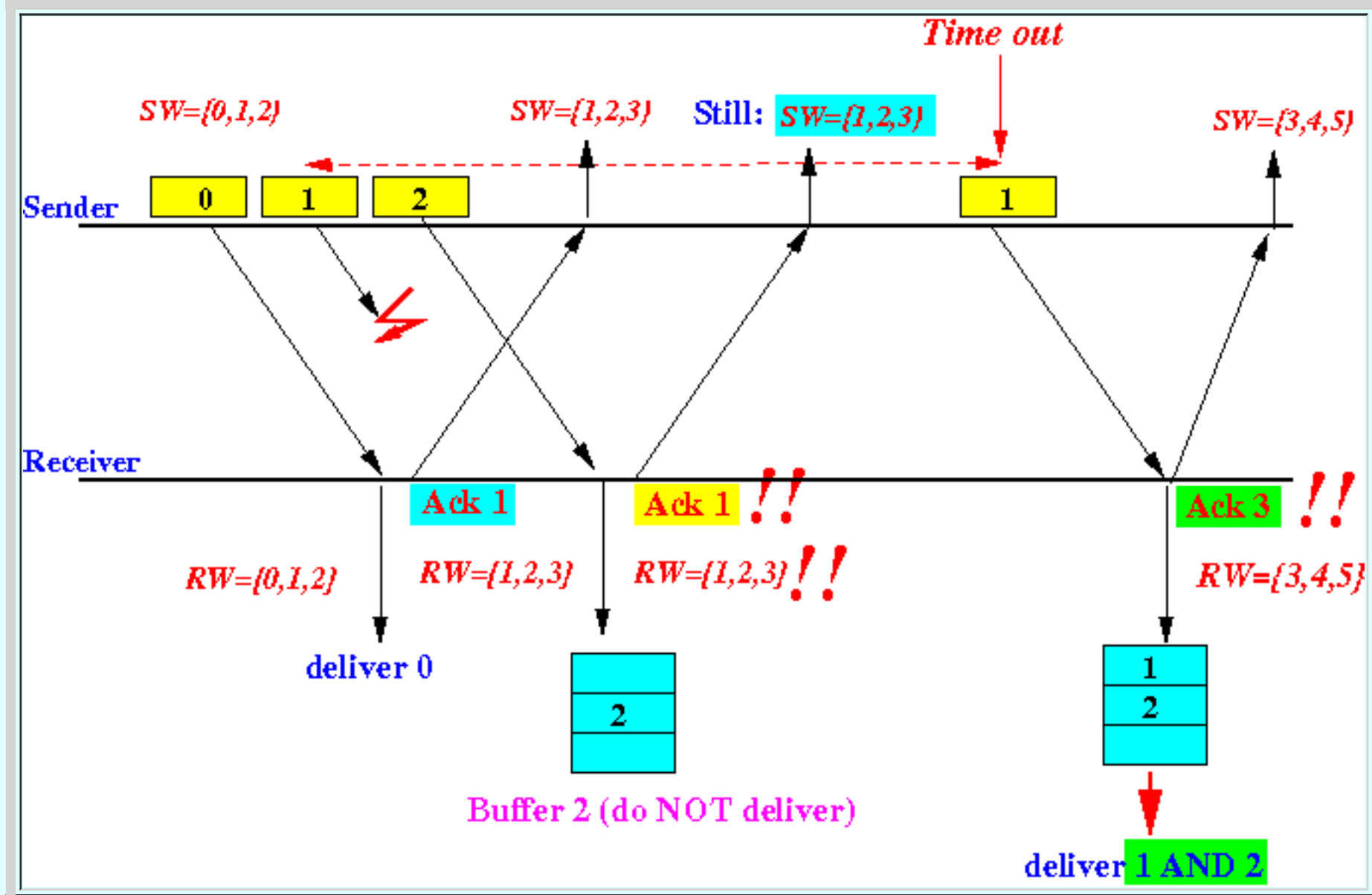
Sliding Window with cumulative ACK: data frame lost

- ***Sliding Window* with Cumulative ACK Example: *Data Frame* Error**
  - **Sliding Window** operation when a ***data* frame** is **corrupted**:

    - **Sender:**
      - Send window = **[0, 1, 2]**

    - **Receiver:**
      - Receive window = **[0, 1, 2]**

    - **Example: data frame 1** is **lost**



Comments:

- **Frame 1** is ***lost***
  - When **frame 2** is **received**, the **receiver** sends back **ACK 1**

    (The **receiver** ***cannot*** send **ACK 3** (to acknowledge **frame 2**), because **ACK 3** will ***also*** acknowledge the ***lost*** **frame 1** !!!)

- **Frame 2** will be **buffered** but **will *not* be *delivered***

- The **sender** will **time out** on **frame 1** and **retransmit** it....

- When **receiver** gets **frame 1**, it will send **ACK 3 !!!**

ACK 3 will **acknowledge** the **data frames 1** and **2** !!!

(Because **frame 2** has **already been received**)

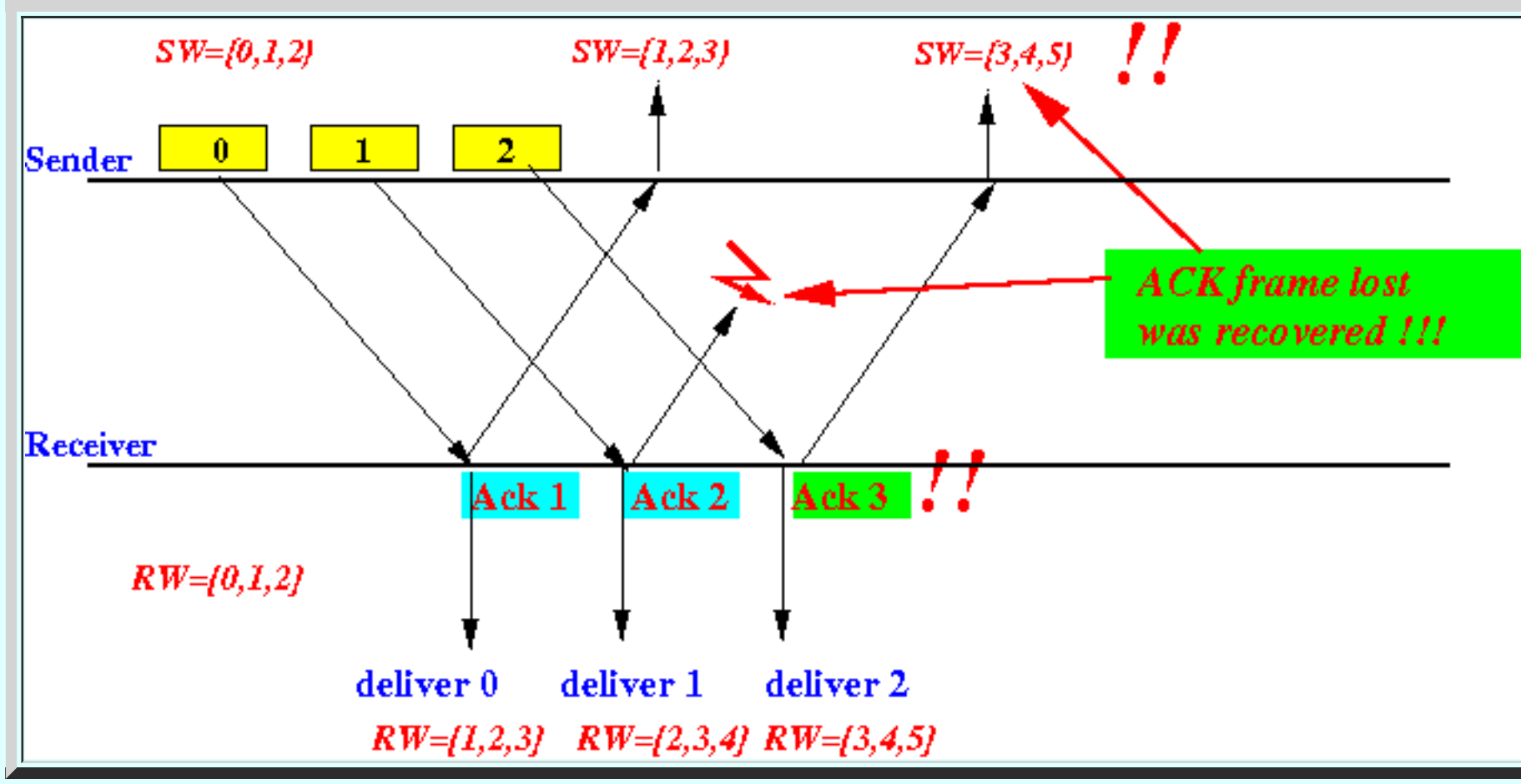
Sliding Window with cumulative ACK: ACK frame lost

- ***Sliding Window* Example: ACK Error - case 1: *not* the last ACK**
  - **Sliding Window** operation when an ***ACK* frame** (not the last one) is **corrupted**:

- **Sender:**
  - Send window = **[0, 1, 2]**

- **Receiver:**
  - Receive window = **[0, 1, 2]**

- **Example: ACK 2** is **lost**



Notes:

- **ACK 2** was **lost**
- When the **sender** receives **ACK 3**:
  - **ACK 3** will ***also*** acknowledge **data frame 1** !!!
    - **ACK 3** will **acknowledge** the **data frame 2** !!!
    - It's like **receiving ACK 2 *again*** !!!

(Here you can see the **ACK protection** mechanism of ***cumulative* ACK in action** !!!!)

- ***Sliding Window* Example: ACK Error - case 2: the *last* ACK is lost**

  - ***Sliding Window*** operation when an ***last* ACK frame** is **lost**:

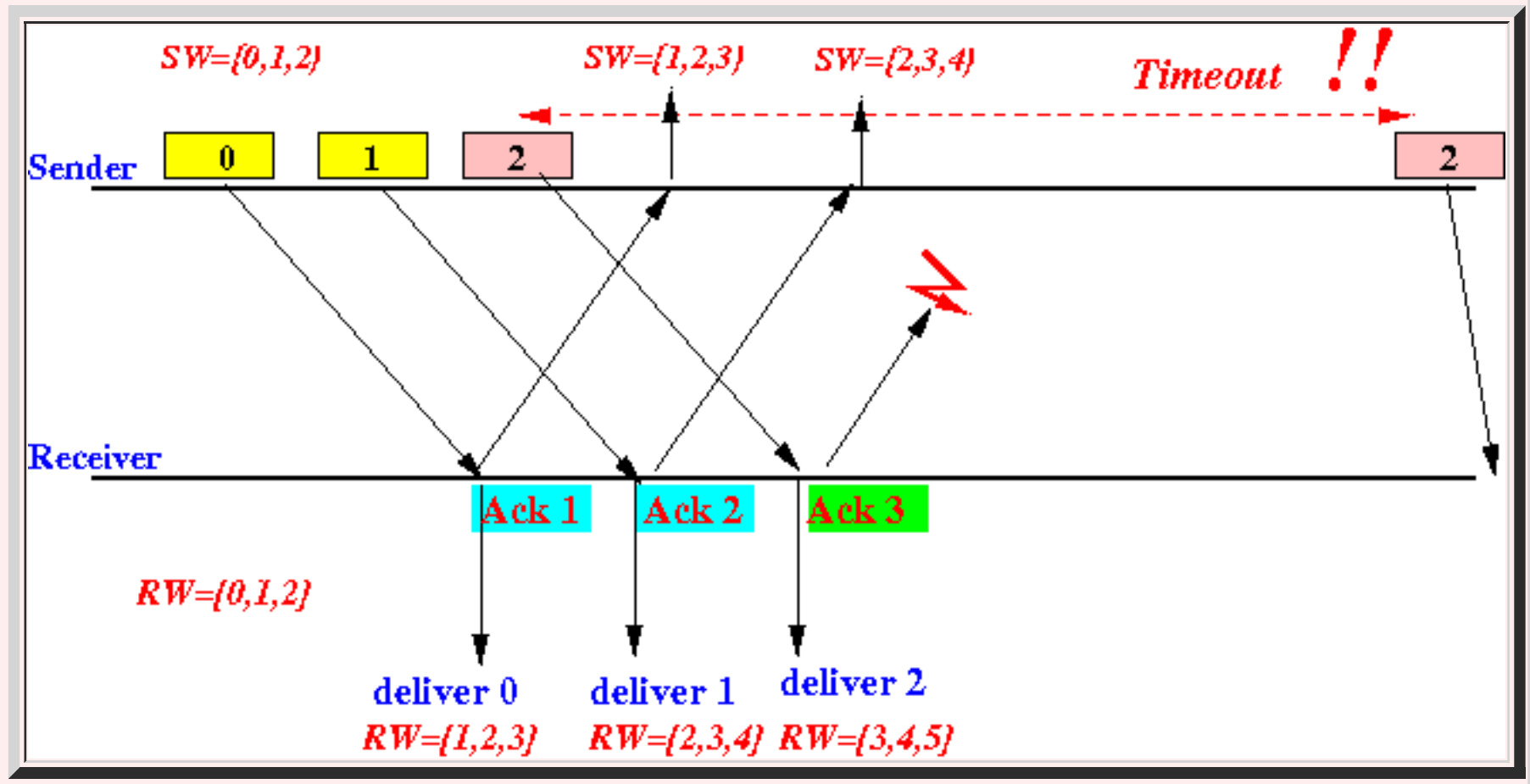
    - **Sender:**

      - Send window = **[0, 1, 2]**

    - **Receiver:**

      - Receive window = **[0, 1, 2]**

    - **Example:**



Explanation:

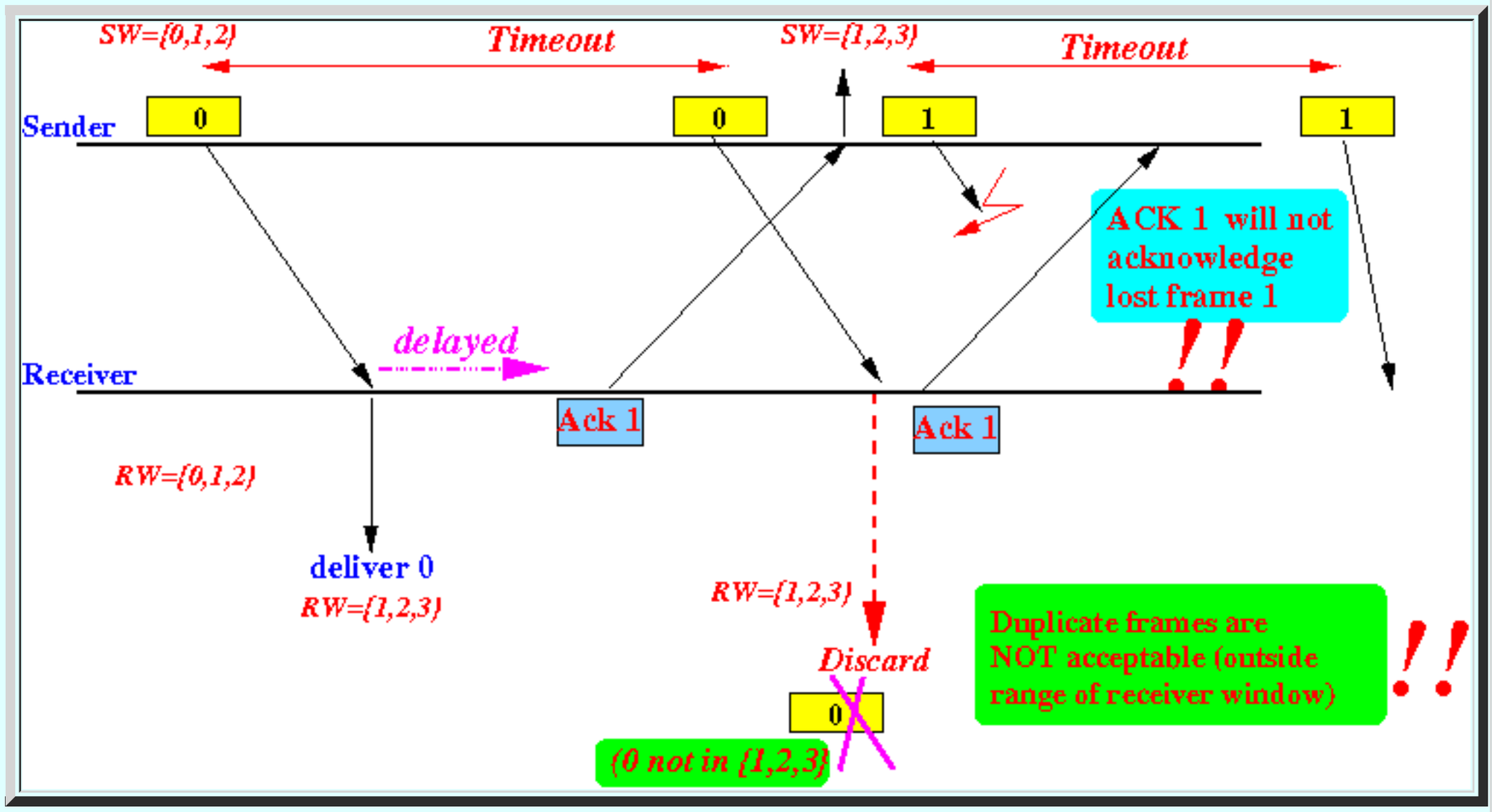
- When the ***last* ACK** is **lost**, the **sender** will **time out** and ***retransmit*** the ***last* data frame**

  (The ***last* ACK** is ***not* protected** by an **later ACK** !!!)

Sliding Window with cumulative ACK: *delayed* ACK

- ***Sliding Window* Example: duplicate (delayed) ACK Error**
  - ***Sliding Window*** operation when an ***ACK* frame** is ***delayed***:

    - **Sender:**
      - Send window = **[0, 1, 2]**
    - **Receiver:**
      - Receive window = **[0, 1, 2]**
    - **Example:**



Notes:

- The **ACK 1** (for **frame 0**) is ***late***
  - **Sender** will **retransmit** the **frame 0**
- The ***duplicate* data frame 0** is ***discarded*** **because**
  - **Receiver window = [1,2,3]**
  - **Data frame seq no = 0**
  - **0 ∉ [1,2,3] !!!**

So these ***sliding* window** will ***ensure***:

- delivering **data frames** exactly ***once*** !!!

Correctness proof of the *Sliding Window* protocol (using *infinite* sequence numbers)

- **Reliable guarantee of the *Sliding Window* protocol**
  - **Claim:**

The ***Sliding Window*** **protocol** using ***infinite*** **set of sequence numbers** can ***guarantee*** that:

- The **receiver** will ***deliver*** each frame ***exactly*** **once**, and
- The **frames** are will be delivered in the **order** that were **transmitted**

Proof: by **showing** the following **3 claims**:

- **Each frame** will be **received** **at least** ***once*** by the receiver.
- A **received frame** will be ***delivered*** **exactly** ***once***
- **Received frames** are **delivered** in the ***transmitted*** **order**

- **Claim 1: Each frame will be received at least once by the receiver**
  - **Claim 1:**

- A **data frame** (say: **frame** ***x***) will be **received** ***at least once*** by the **receiver**.

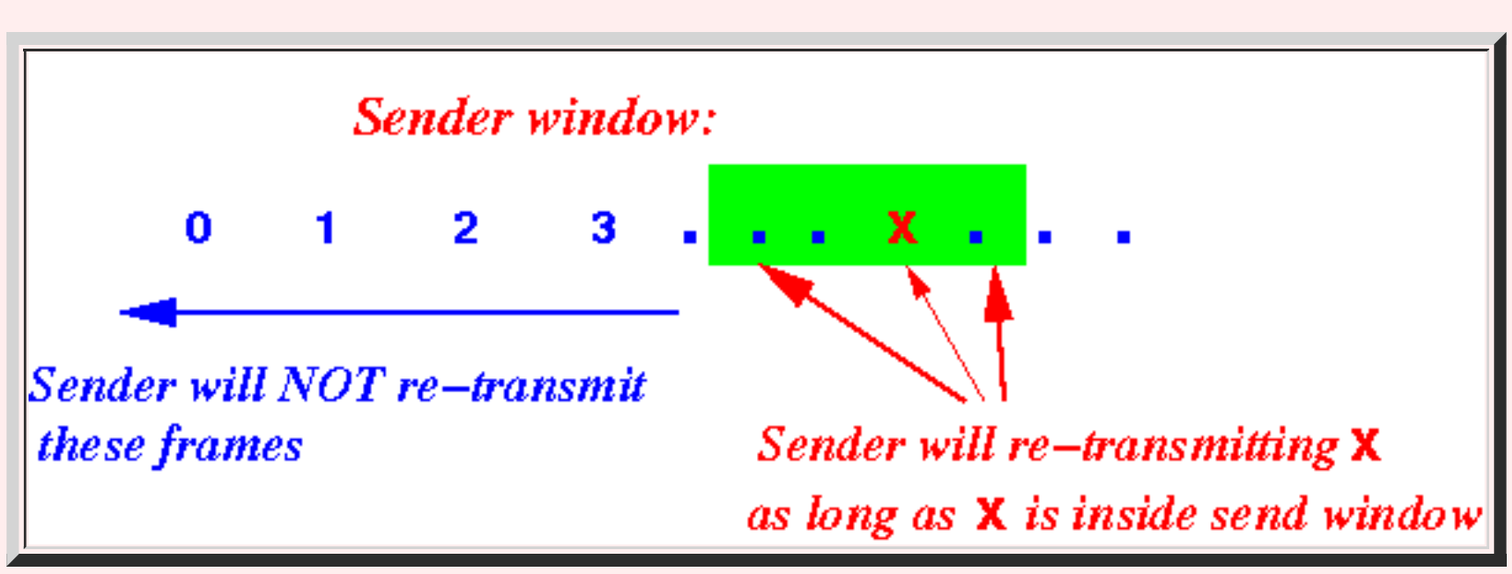
Proof:

- **Fact 1:**

  - As long as **data frame** ***x*** is ***outstanding*** **(= unacknowledged)**, the **sender** will **keep transmitting** the **data frame** ***x***.

    (**Sender** can ***timeout*** on an ***outstanding*** **frame** and ***re-transmit***)

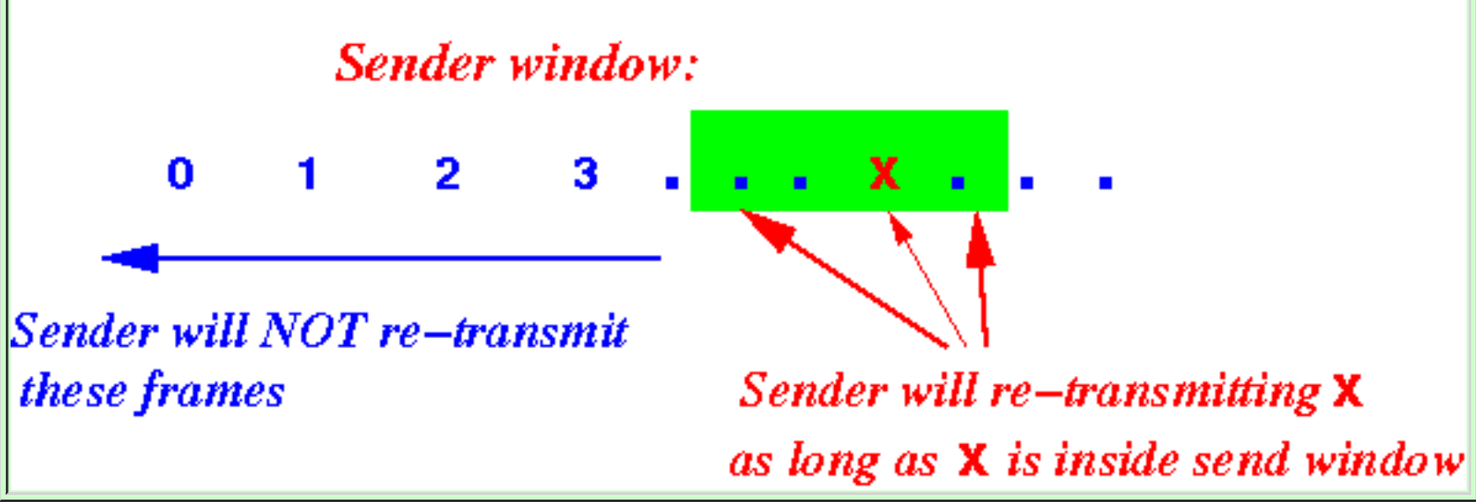
  - **Rephrased:** as long as ***x*** is **inside** the ***sender*** **window**:



  the **sender** will **keep sending** the **data frame**

Therefore:

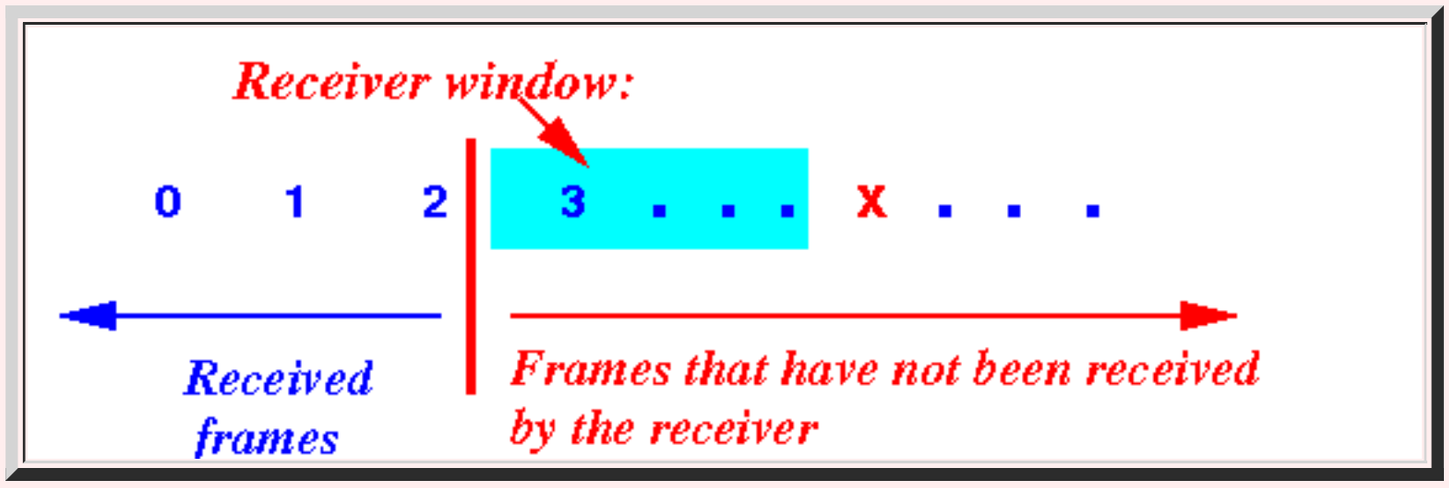
- As long as **data frame** ***x*** is **inside** the ***sender*** **window**:



the **receiver** will have the **opportunity** to **receive** the **data frame *x***

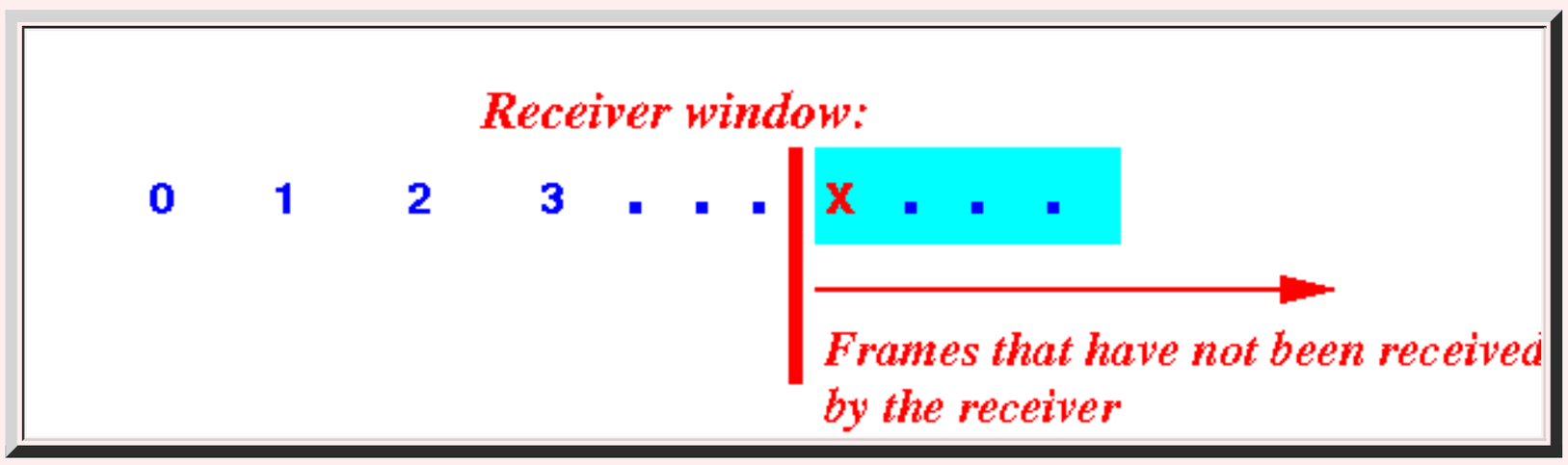
- **Fact 2:**

  - A **data frame *x*** that was ***not* received** by the **receiver** will be ***inside*** the **receiver window** or ***pass*** the **receiver window**:



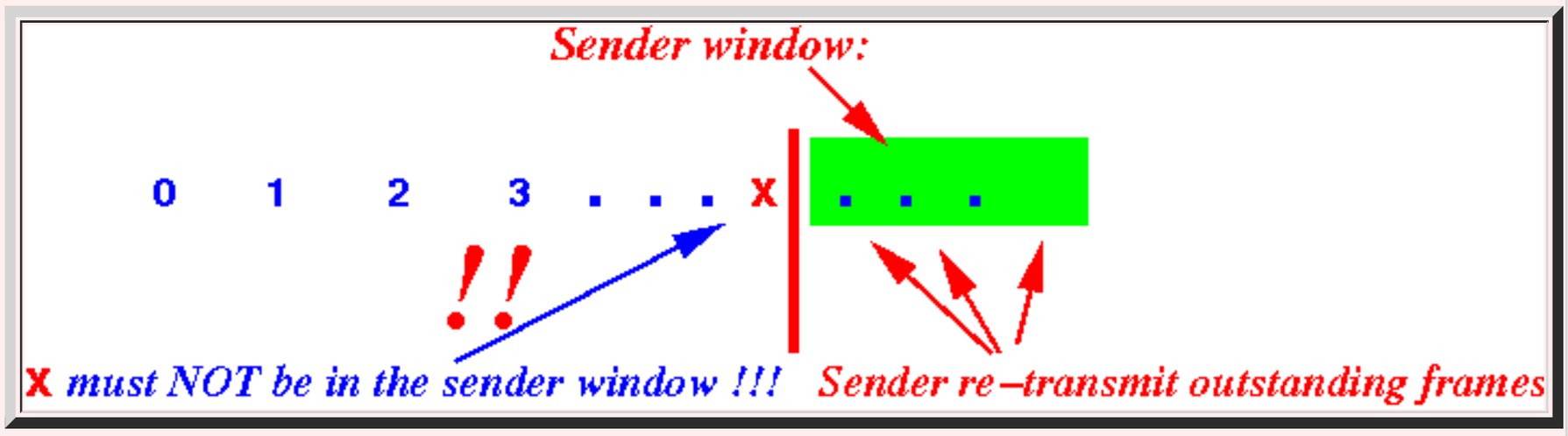
- **Therefore**, the **receiver** will ***not* receive** a **frame *x*** ***if***:

1. The **receiver** has ***not* received** the **frame *x*** **yet**:



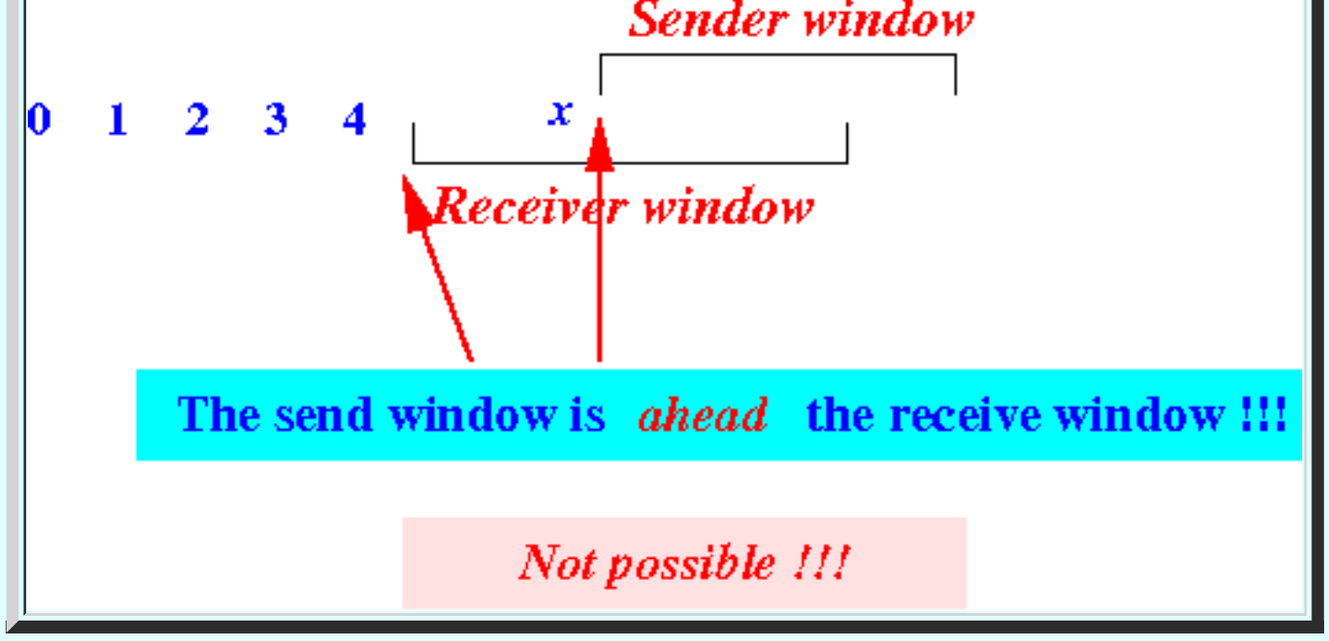
---

2. **And** the **sender** will ***not* transmit** the **frame *x*** **any more**:



I.e.: **frame *x*** must **not** be **inside** the **send window**

In other words:



And this **situation** is ***impossible*** **because** we have **shown** that:

- The **receive window** will ***always*** be **lined up *or* ahead** the ***send*** **window** !!!

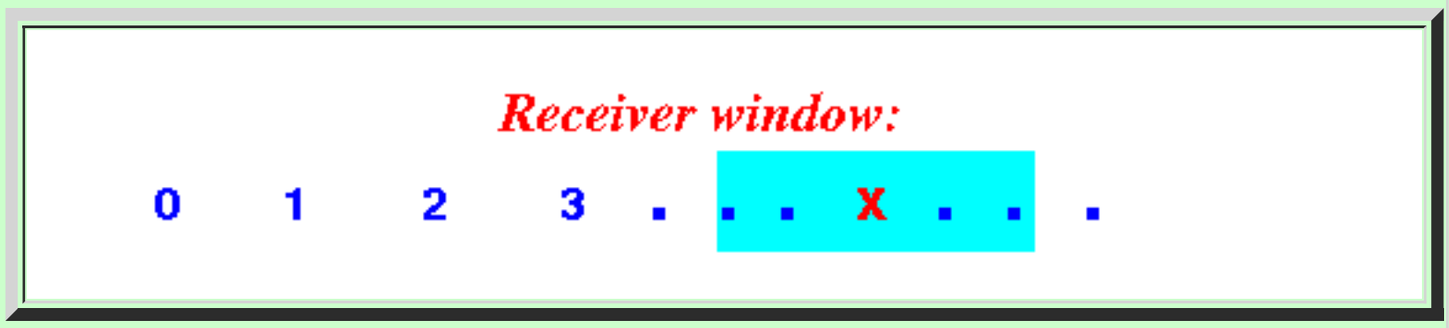
  See: **click here**

- **Claim 2: A received frame will be delivered exactly once**
  - **Claim 2:**

    - A **frame *x*** will be ***delivered*** exactly ***once***

    Proof:

    - **While** the **frame *x*** is ***inside*** the **receiver window**:

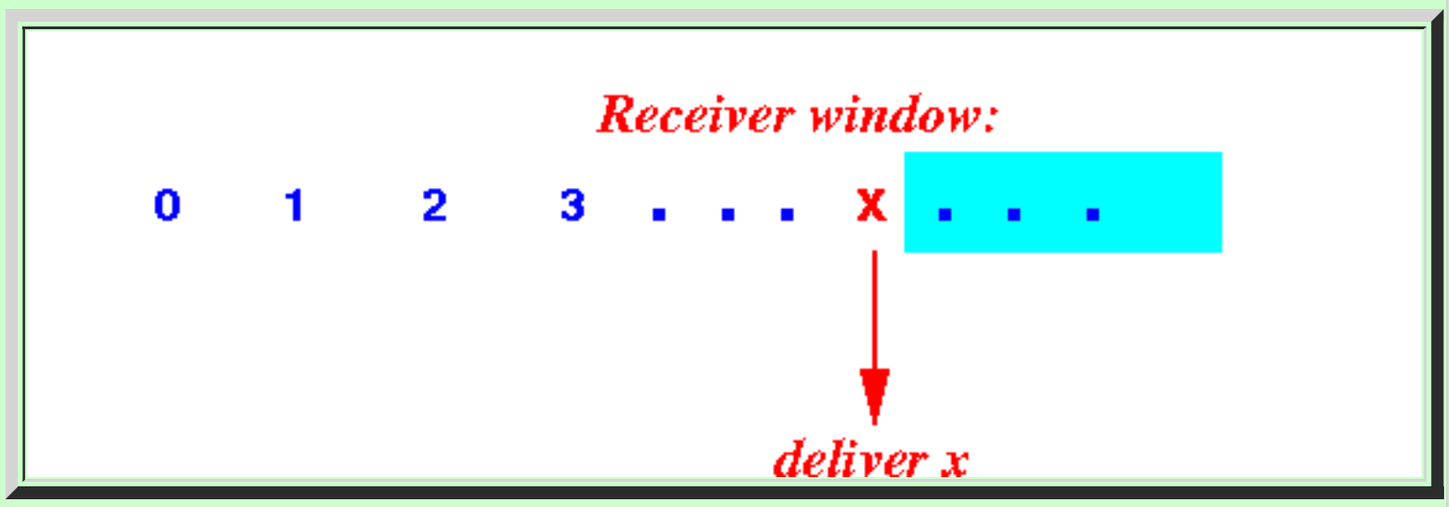
      

      the **receiver** can receive ***duplicate*** **frame *x***

    - Because ***each*** **frame** has a ***unique*** **sequence number**:

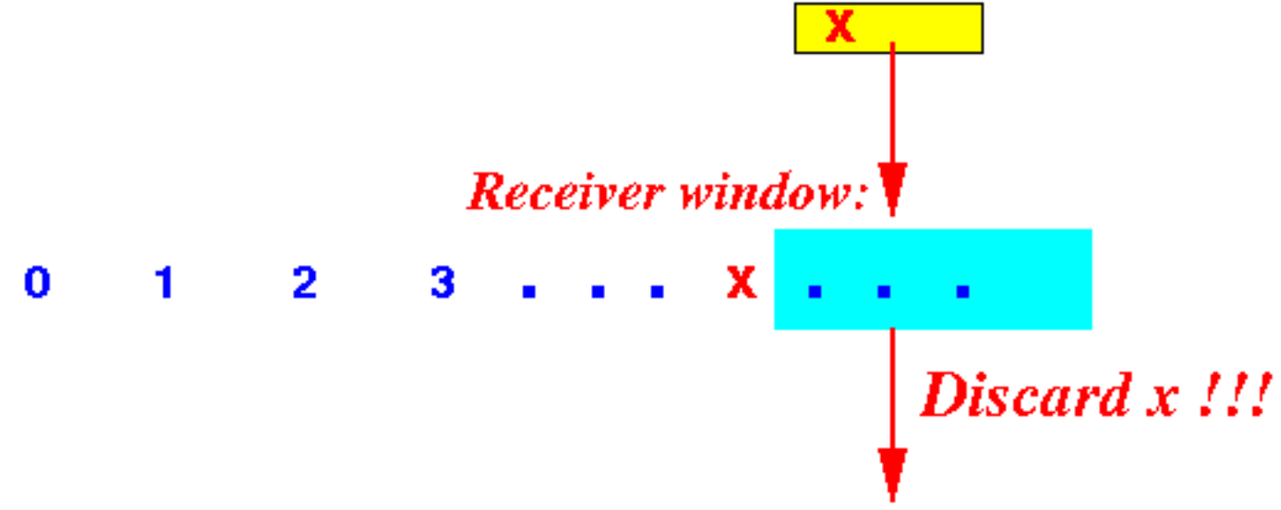
      - The **receiver** can ***remove*** the **duplicates** !!!

    - **When** the **receiver *delivers*** the **data frame *x***, the **receiver window** will:

      also *slide* the **receiver window pass the value *x*** !!!

    - In **this state**:



Retransmission of **data frame *x*** are **discard** !!!

- **Therefore:**

  - A **data frame *x*** will be **delivered *exactly* once** !!!

- **Claim 3: Received frames are delivered in the transmitted order**
  - This **cliam** is **obvious**:

    - **Data frames** are ***transmitted*** in ***increasing* order** by the **sender**

    - **Data frames** are **delivered** in ***increasing* order** by the **receiver**

      - There is ***only* one ordering** possible with a **set** of **integer values** that is ***increasing* order**

        (**Namely: x**, **x+1**, **x+2**, ...)

    - **Therefore:**

      - **Received frames** are ***delivered*** in the **transmitted order**

Problem caused by finite sequence numbers - ambiguity

- **Prelude: Acknowledgement scheme used in the discussion**
  - **Fact:**

    - The **discussion** on the **correctness** of the **sliding window algorithm** does **not** depend on the **ACK scheme** used

    - **So**, for **simplicity**, I will **use** ***selective*** **acknowledge**

      I.e.:

      ```
      ACK n   ===>  ACK data frame n only
      ```

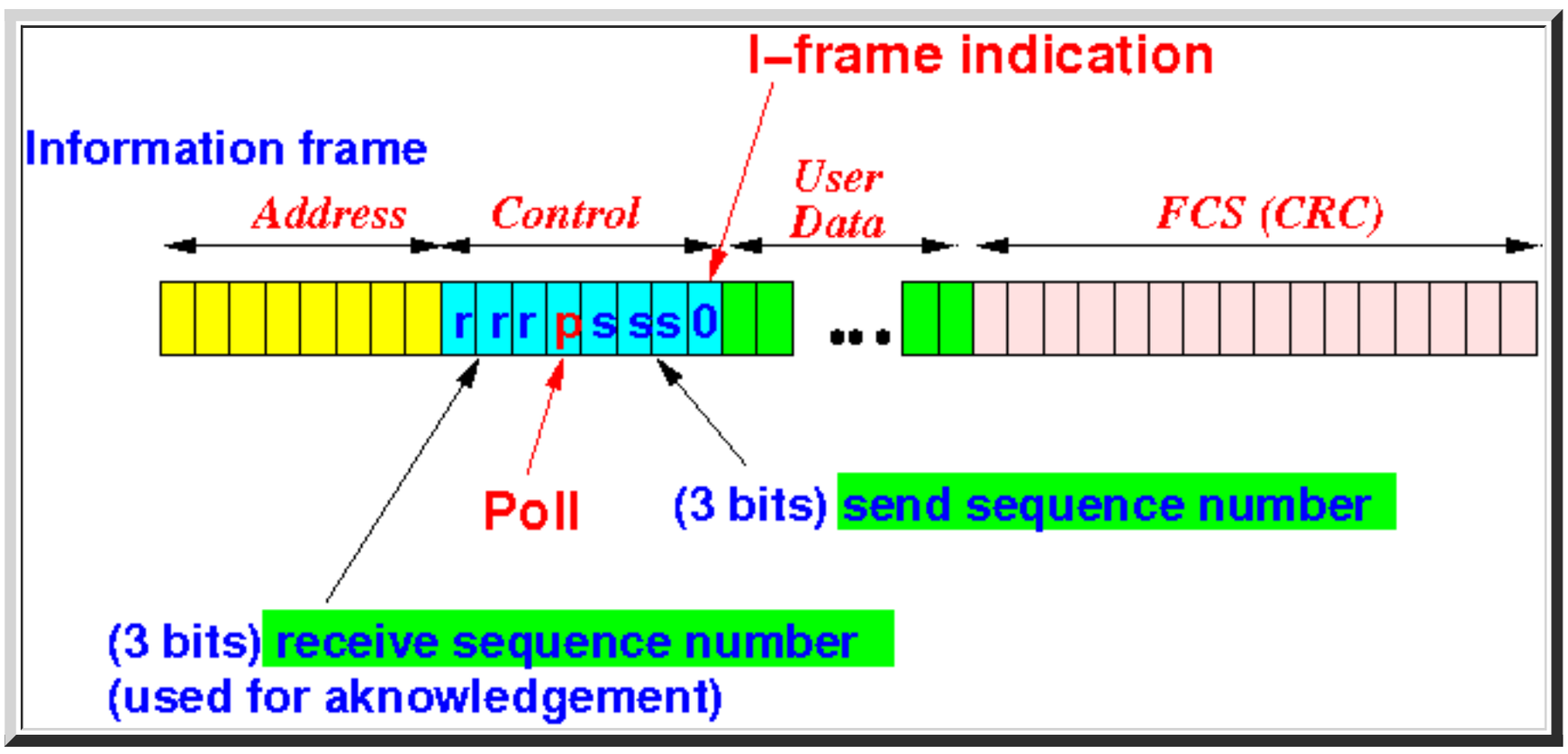
- **Sequence numbers in frames**
  - **Fact:**

    - **(Send/Receive) Sequence numbers** are found in the

      - **Header** portion

      of **data/ACK frames**

  - **Example:** **HDLC I-frame**



  - **Sequence numbers** used in **data communication**:

    - **Sequence numbers** is **always** a **binary field** of ***n*** **bits**

    - The ***number*** of **(different) sequence numbers** available = 2^n

  - **Example:**

    - **Suppose** a **frame header** uses **2 bits sequence numbers**:

      ```
        Frame header      Body (data)         CRC
      +--------------+--------------------+-----+
      ```

```
|x|x|   ....   |       .....          | ... |
+-------------+----------------------+-----+
Seq.
number
```

- The **total number** of ***sequence numbers*** = **4**

  Namely:

  - **00, 01, 10, 11**

  Or:

  - **0, 1, 2, 3** (in **decimal base notation**)

- **The effect of *finite* number of sequence numbers**
  - **Effect:**
    - ***Different* data frames** will be **identified** using the ***same* sequence number**
  - **Example:**
    - **Suppose** the **available sequence numbers** are:
      - **00, 01, 10, 11**
        Or: 0, 1, 2, 3

    - **Frames** will be **identified** with **sequence numbers** as **follows**:

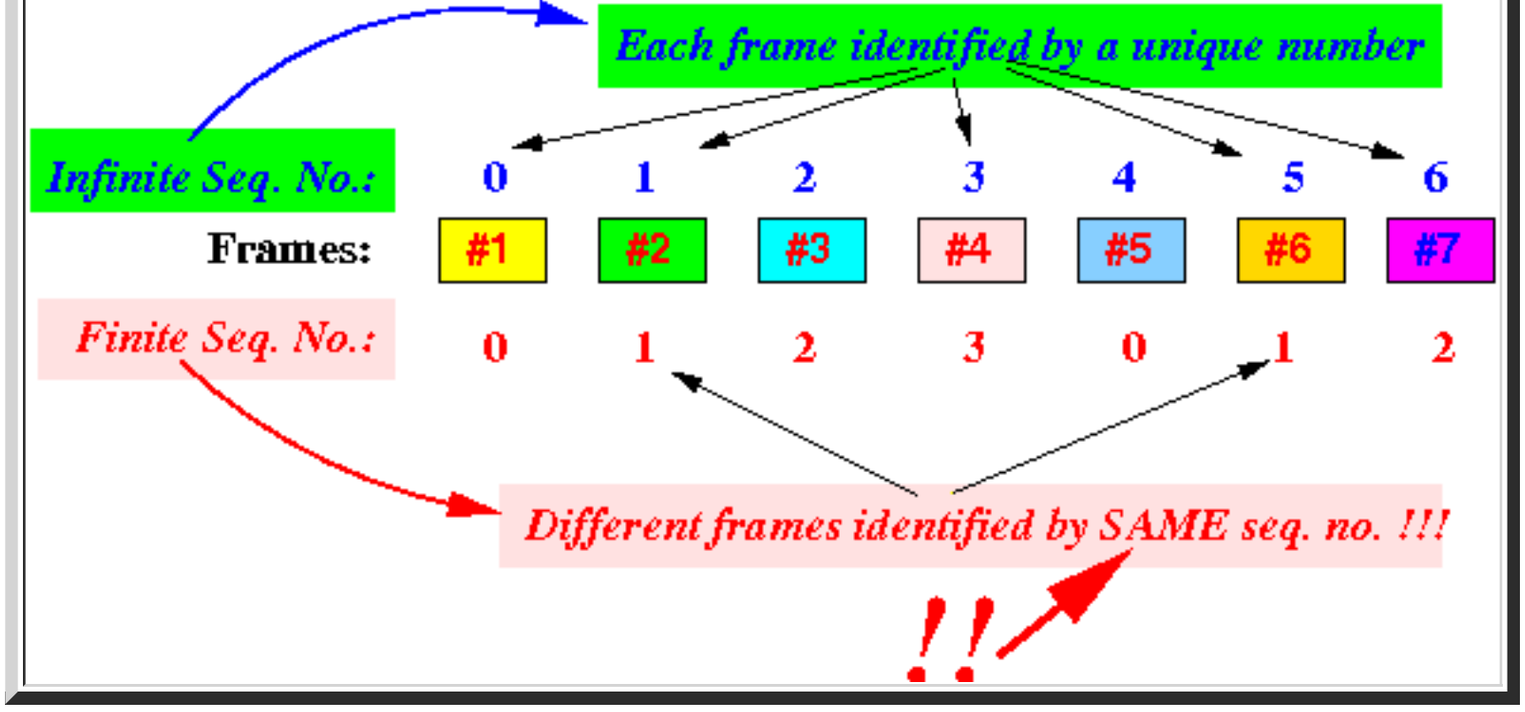
```
Frames:          #1   #2   #3   #4   #5   #6   #7   #8   #9   #10

Infinite
seq. no.:         0    1    2    3    4    5    6    7    8    9

4 seq no:        00   01   10   11   00   01   10   11   00   01 ...
          or:     0    1    2    3    0    1    2    3    0    1  ...
                       ^                   ^
                       |                   |
                  Frame #2 and #6 are both identified with
                  the seq. no. 01
```

      Notice that the **sequence numbers** will **repeat**

  - **Comment:**
    - I will use **decimal sequence numbers** for **convenience**....

      (Remember they are **binary numbers** in the **frame header**)

- **Problem caused by *finite* sequence numbers: ambiguity...**
  - **Problem** created by a ***finite* number** of **sequence numbers**:



Explanation:

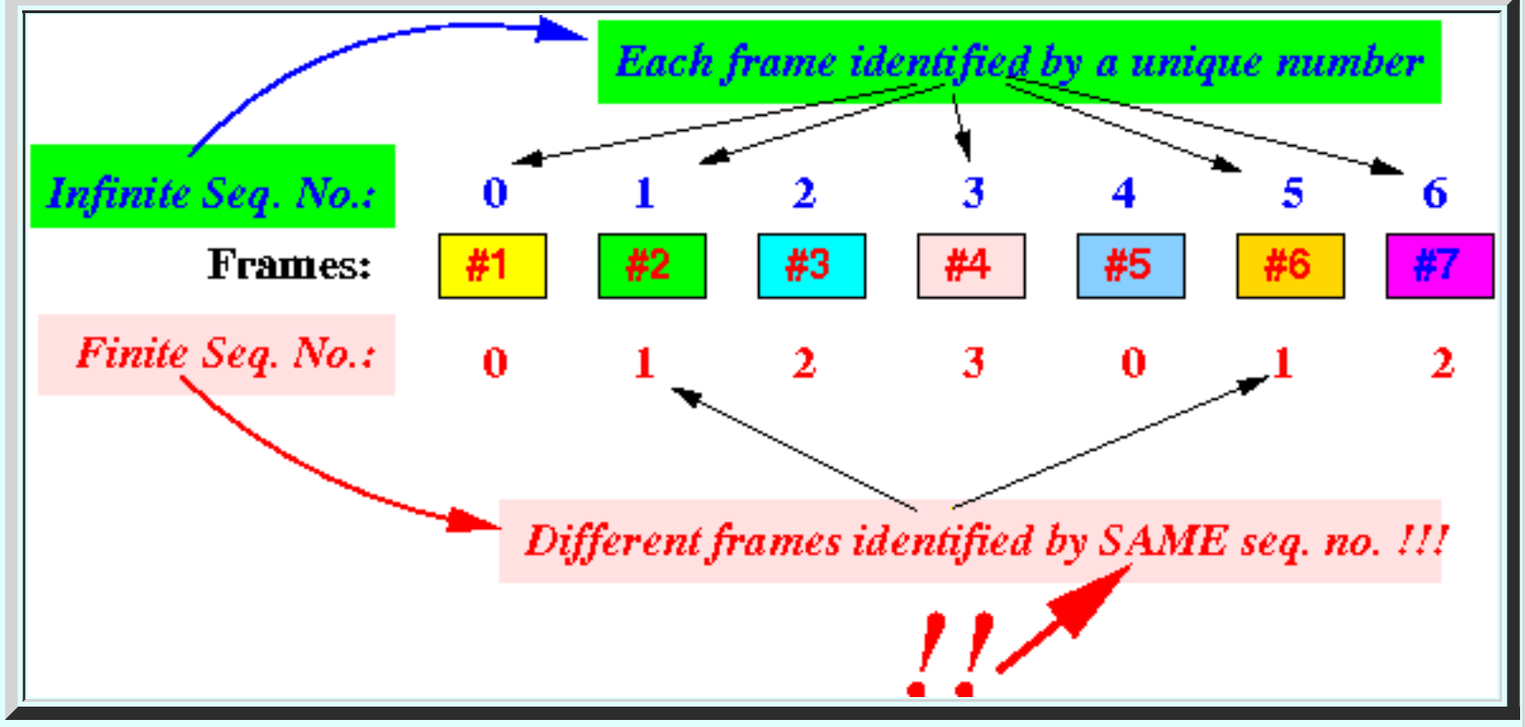
- The **sender** and **receiver** must ***re-use*** the **sequence numbers** to number ***different*** **frames**

  (Sooner or later, the **sender** and **receiver** will **run out** of **sequence numbers** and must **re-use (= repeat)** them again.)

- **Result:**

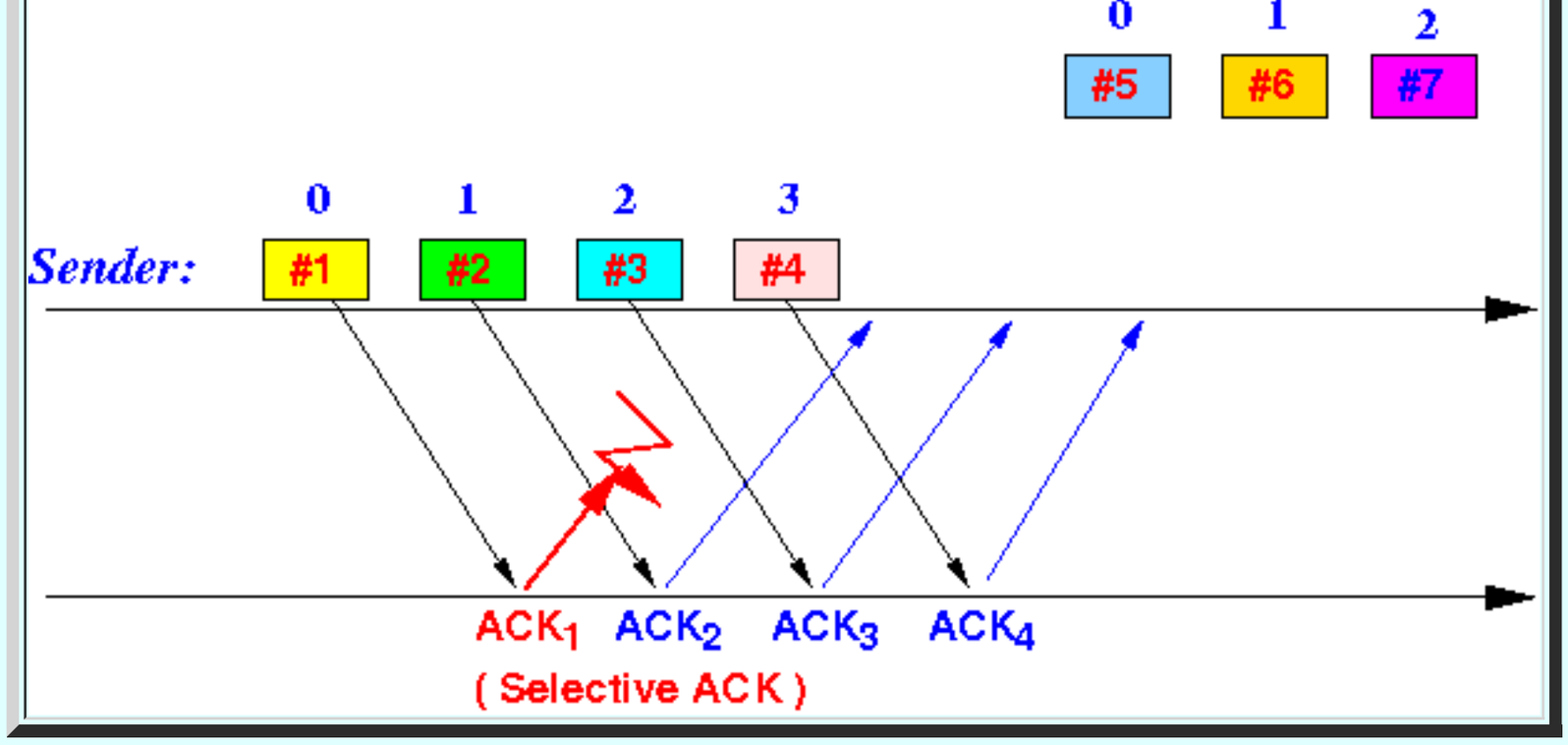
  - The ***same*** **sequence number** is used to **identified** ***different*** **frames**

Consequently: ***ambiguity***



- The **receiver** can **not** tell **frame #2** from **frame #6** by the **sequence number 1** !!!

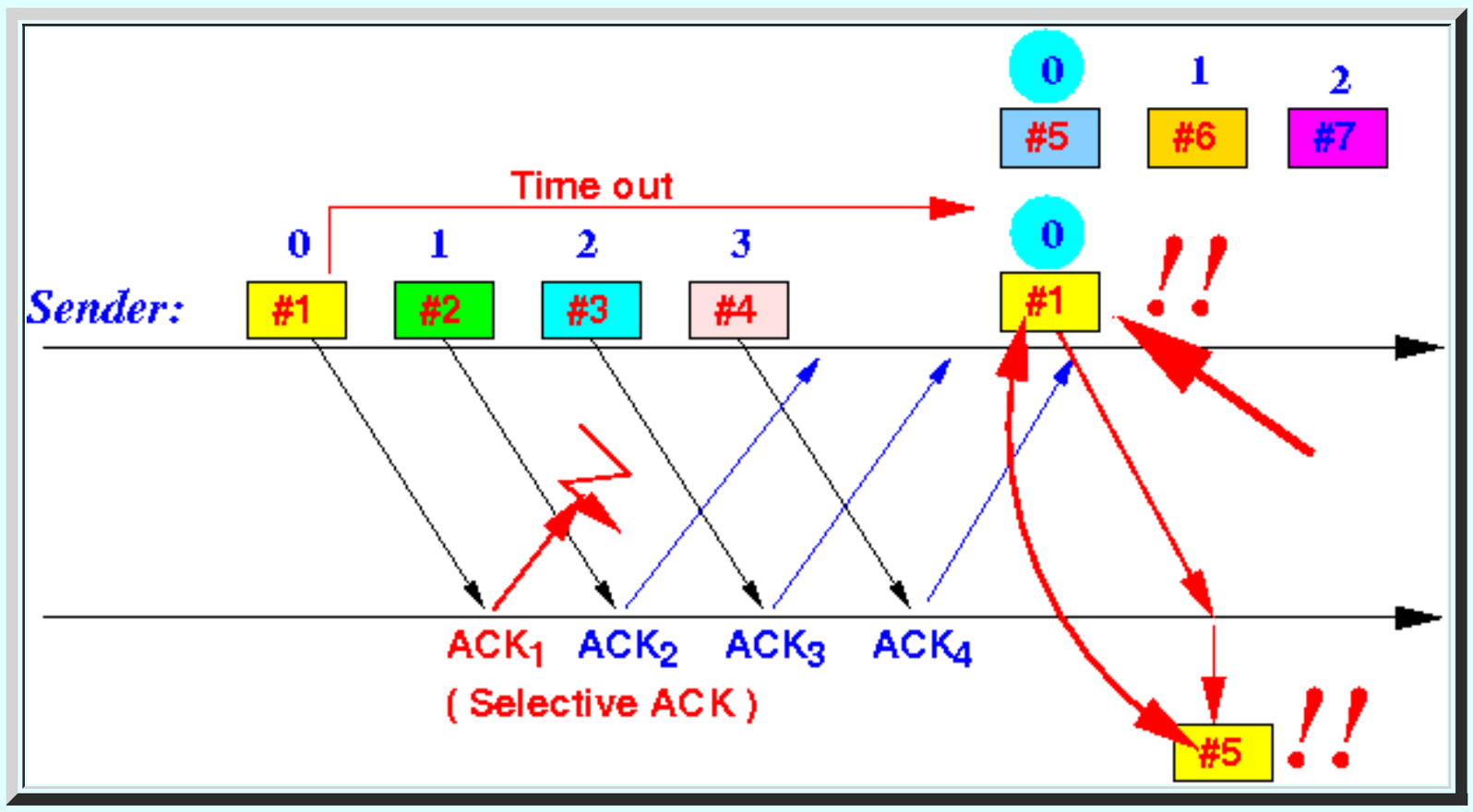
- **How can ambiguity cause problems**
  - **Potential problem scenario**:
    - The **sender** transmits **data frame 0** using **send seq no 0**:



But:

- The **ACK** for **data frame 0** is ***lost*** !!!

- **Later**, the **sender** times out and ***re-transmits*** the **data frame 0** (with **send seq no 0**):



It is **possible** that:

- The **receiver thinks** it was **(new) data frame #5** that was **transmitted** using **send seq no 0** !!!

Result:

- The **receiver** will ***deliver*** a ***re-transmitted*** **(old) frame *more* than once** !!!

- **Quiz: is frame numbered with seq no 0 the frame #1 or frame #5 ???**
  - **Operational parameters:**
    - **Sender** and **receiver** use **2 bits** sequence numbers
      - **Sequence numbers** = **0, 1, 2, 3**

      The **data frames** are **identified** as **follows**:

```
Frame:      #1   #2   #3   #4   #5   #6   #7   #8   #9   .....
```

```
Seq. No.:    0   1   2   3   0   1   2   3   0   .....
```

Notice that:

- **Frame #1** and **frame #5** are **both identified** by **sequence number 0** !!!

- **Suppose:**

  - **Send window size** = **3**
  - **Receive window size** = **3**

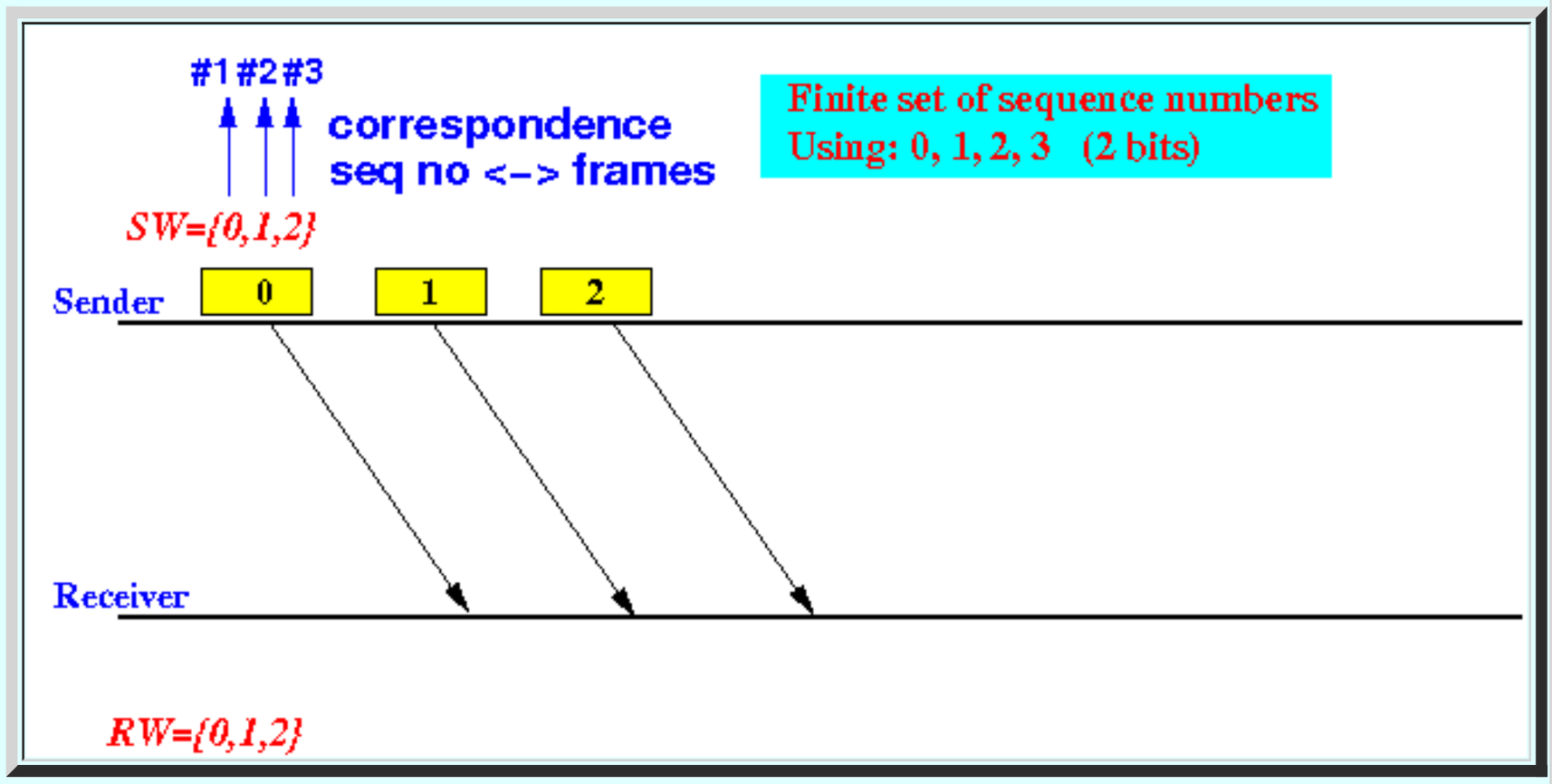
- **Consider** the following ***scenario***:

  - **Sender** sends **3 frames (#1, #2, #3)** (due to **send window size = 3**, the **sender cannot** send **more** !!!)

    The **frames** are identified with **sequence numbers 0**, **1** and **2** to the **receiver**

    - **All frames** has been **received** (correctly)

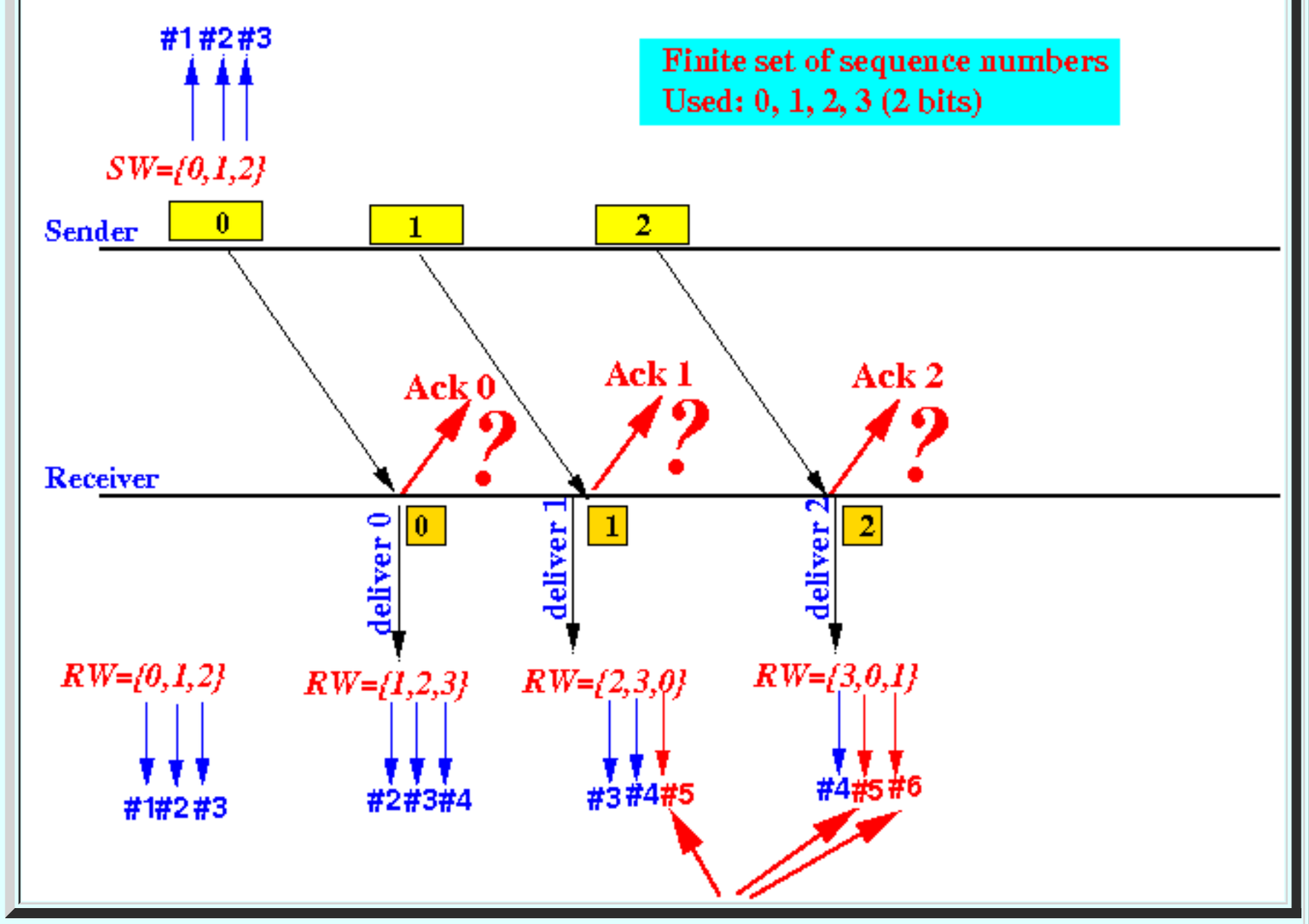
Graphically:



- **Receiver** sends **ACK_0**, **ACK_1** and **ACK_2**

  - The **fate** of the **ACK frames** are ***unknown***

  Graphically:



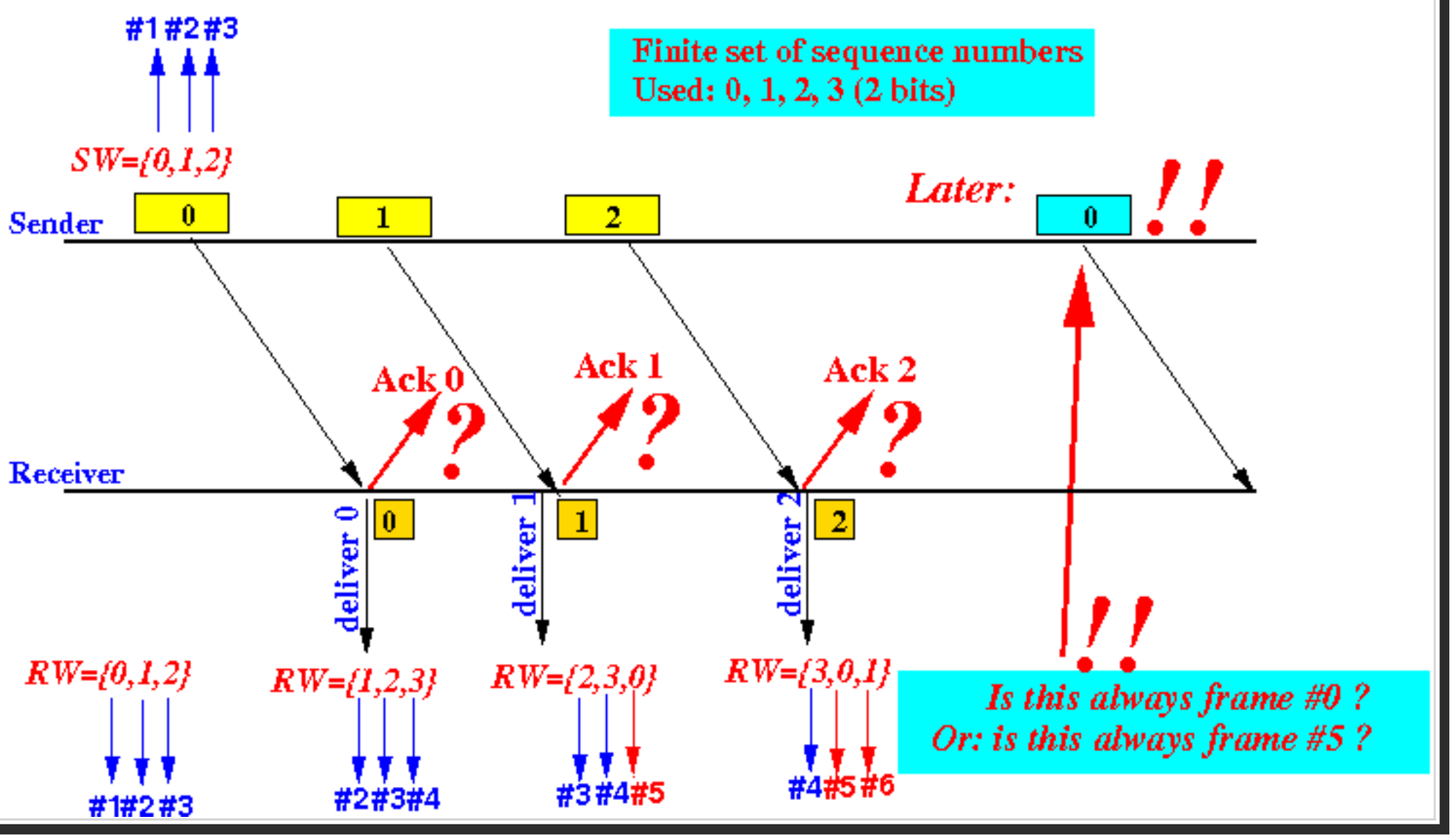
Notice that:

- **Receiver window** = **[3,0,1]**

  Furthermore:

  - The **seq. no. 0** is used to **identify** the **data frame #5** !!!

- **Suppose** the **receiver** received a **data frame** with **sequence number = 0**:



$64,000 **question:**

- Is the **frame** **identified** by ***sequence number*** **#0**:

- **Always correspond** to the ***frame #1*** ?? **or**
- **Always correspond** to the ***frame #5*** ??

- **Comment:**

  - ***Because*** the **receive window** is **equal** to **[3,0,1]**:

    - The **receiver** is using **seq no "0"** to **identify** the **data frame #5**

    Therefore:

    - The **receiver** will **always *interpret*** that the ***second*** **frame** identified by the **sequence number 0** is the **frame #5**

- The **$64,000 question re-phrased**:

  - Is the ***second*** **frame 0**:

    - ***always*** **(i.e., for *sure*)** the **frame #5** from the **sender ??????**

  - **Answer:**

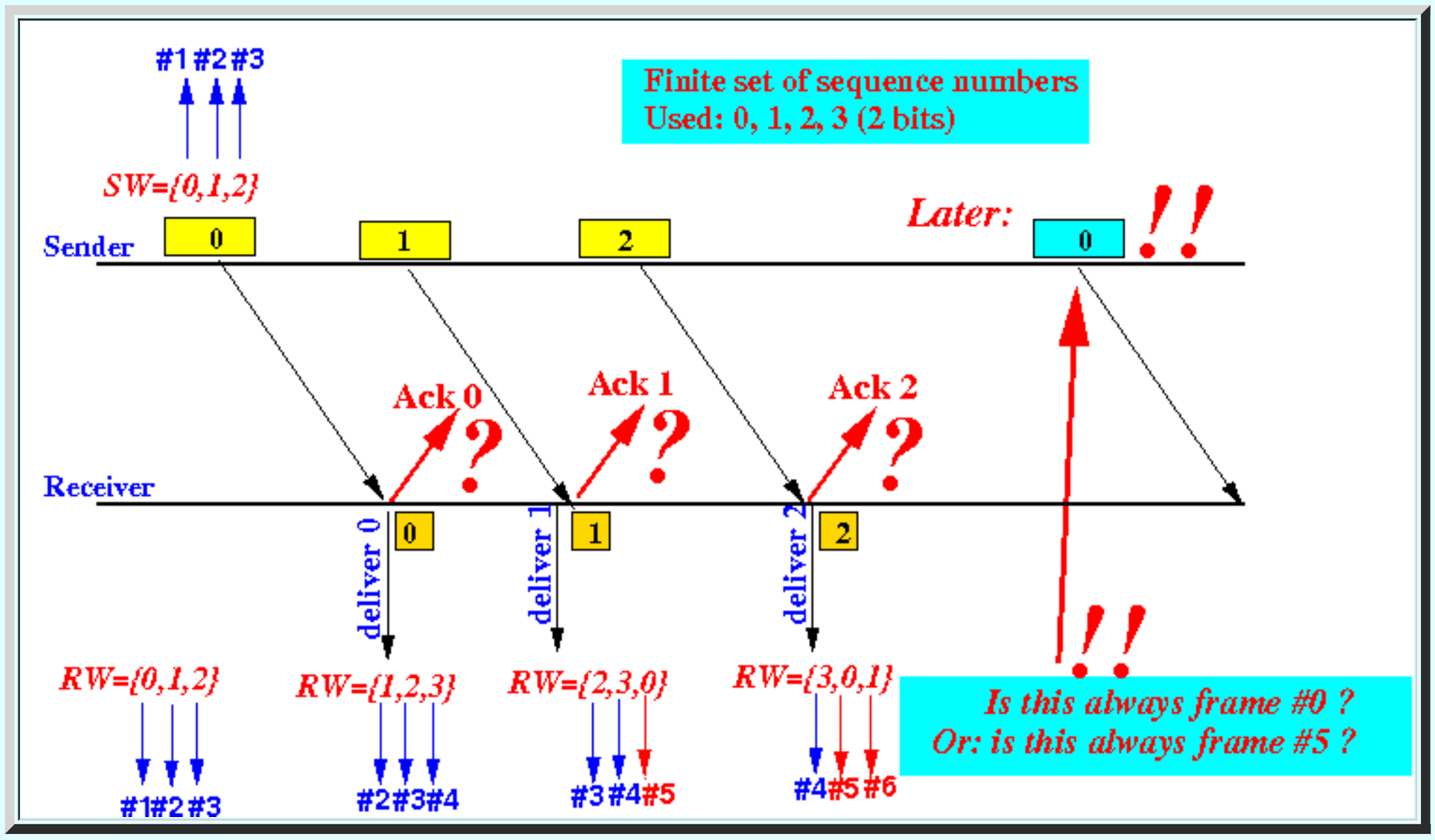
    - ***NO*** **!!!!**

    I will show you **2 scenarios** that that shows that ***both*** **cases** are **possible** !!!

- **Scenario 1: the *second* frame with sequence number 0 is the frame #5**

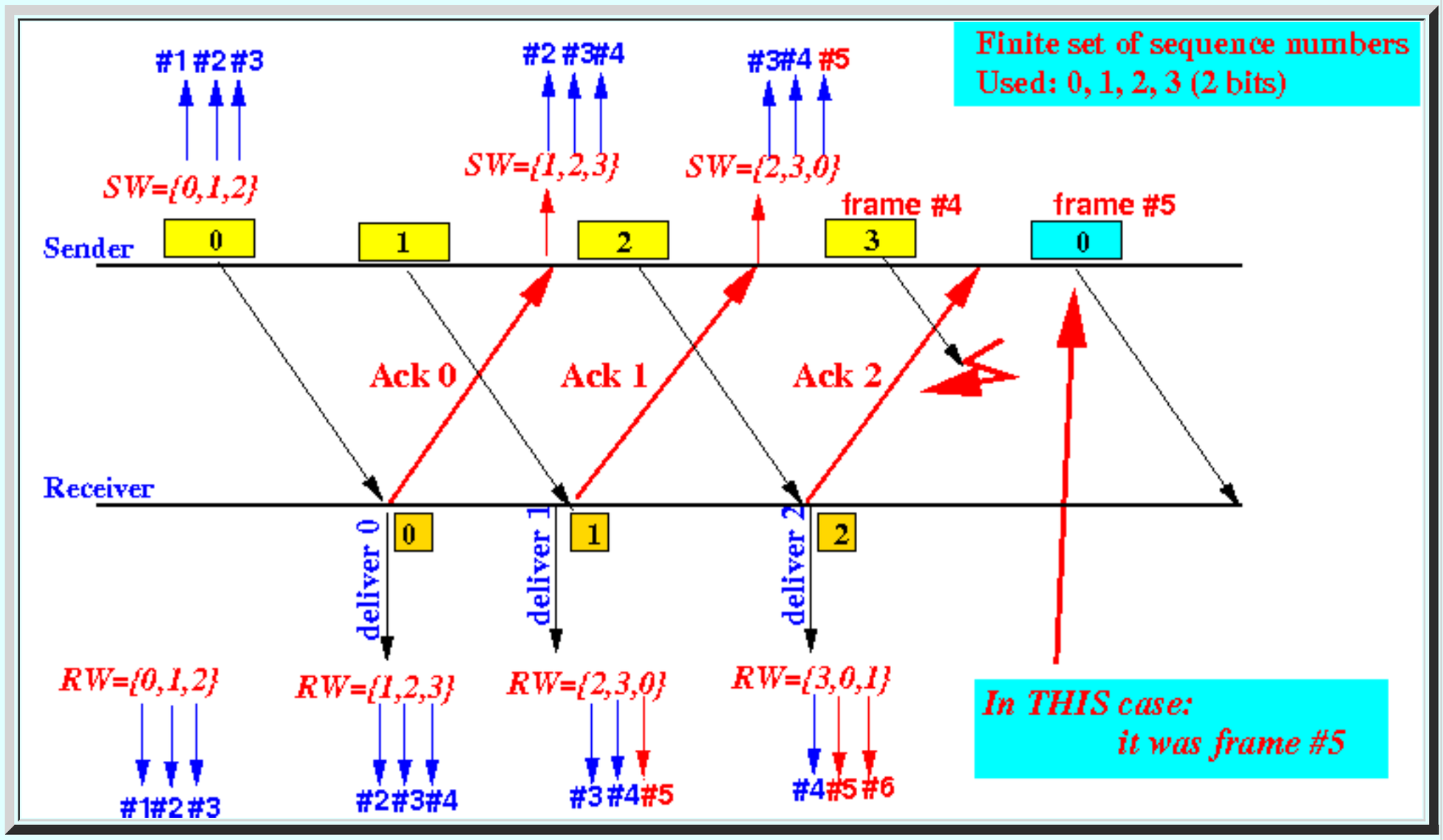
  - **Scenario 1:**

    - **Initial state:**



- **Suppose *all* ACK frames** were **received *correctly***

  And: ***frame 3 (= #4)*** *is* ***lost****:*

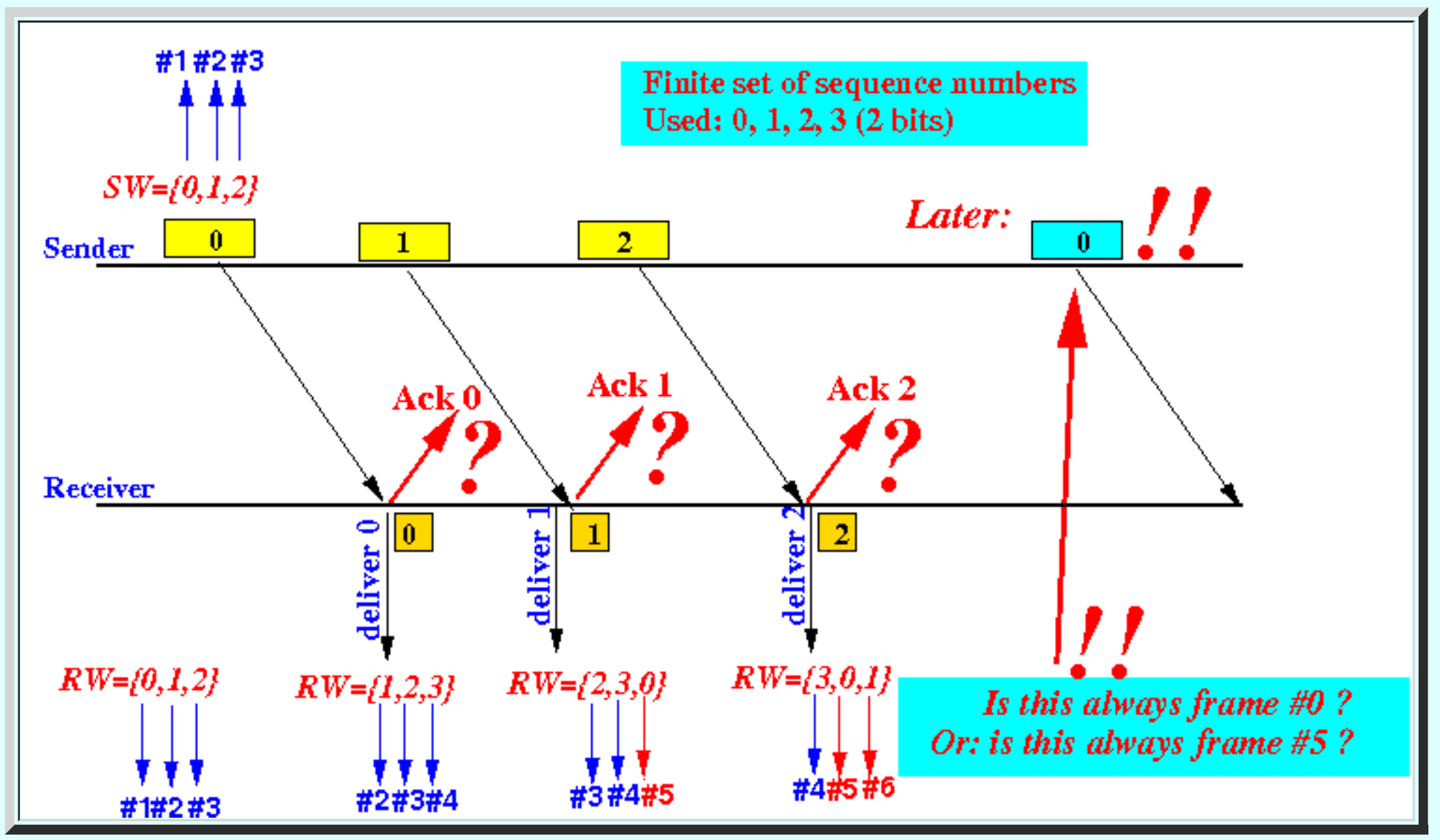


Then:

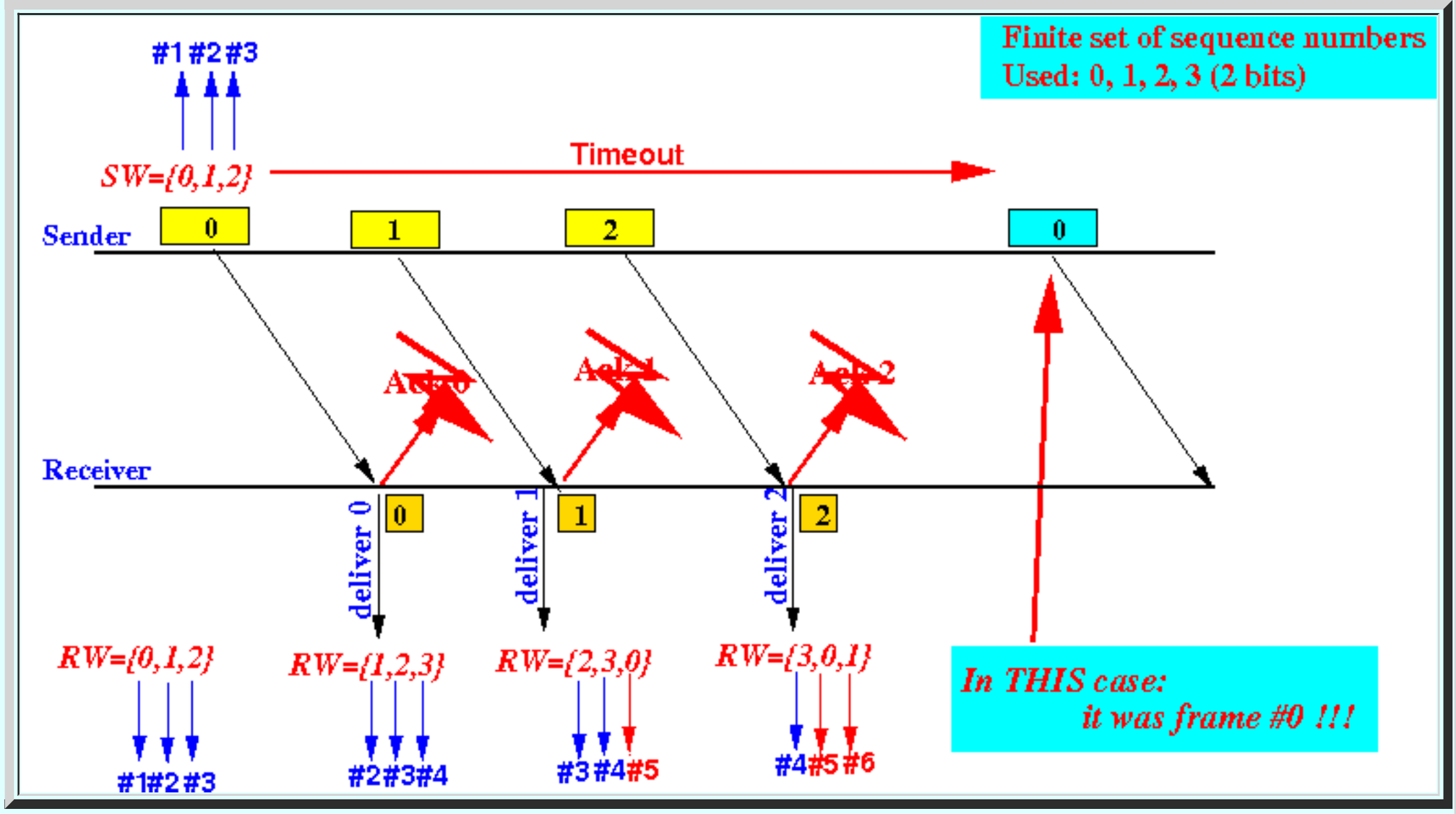
- The ***second*** frame **identified** by **sequence number 0** is **frame #5**

- **Scenario 2: the *second* frame with sequence number 0 is the frame #1**
  - **Scenario 2:**

    - **Initial state:**



- **Suppose *all* ACK frames** were ***lost***:



Then:

- The ***second*** frame **identified** by **sequence number 0** is **frame #0** !!!

- **Conclusion:**

  - The **receiver** does **not** have **sufficient information** to **distinguish** these **2 case** !!!

  - The **sliding window algorithm** will **fail** to **ensure reliable communication** !!!

- **Can we avoid ambiguous situations ?**

  - **Answer:**

    - **Yes** !!!

See **next webpage** !!!

Finite sequence numbers do not always cause *ambiguity*

- **Example of a scenario wthere there is *no* ambiguity**
  - **Operational parameters:**

    - **Sender** and **receiver** use **2 bits** sequence numbers

      - **Sequence numbers** = **0, 1, 2, 3**

    The **data frames** are **identified** as **follows**:

    ```
    Frame:       #1   #2   #3   #4   #5   #6   #7   #8   #9   .....
    Seq. No.:     0    1    2    3    0    1    2    3    0   .....
    ```

    Notice that:

    - **Frame #1** and **frame #5** are **both identified** by **sequence number 0** !!!

    - **Suppose:**

      - **Send window size** = **2** (in previous example, it was **3**)
      - **Receive window size** = **2** (in previous example, it was **3**)

  - **Consider** the following ***scenario***:

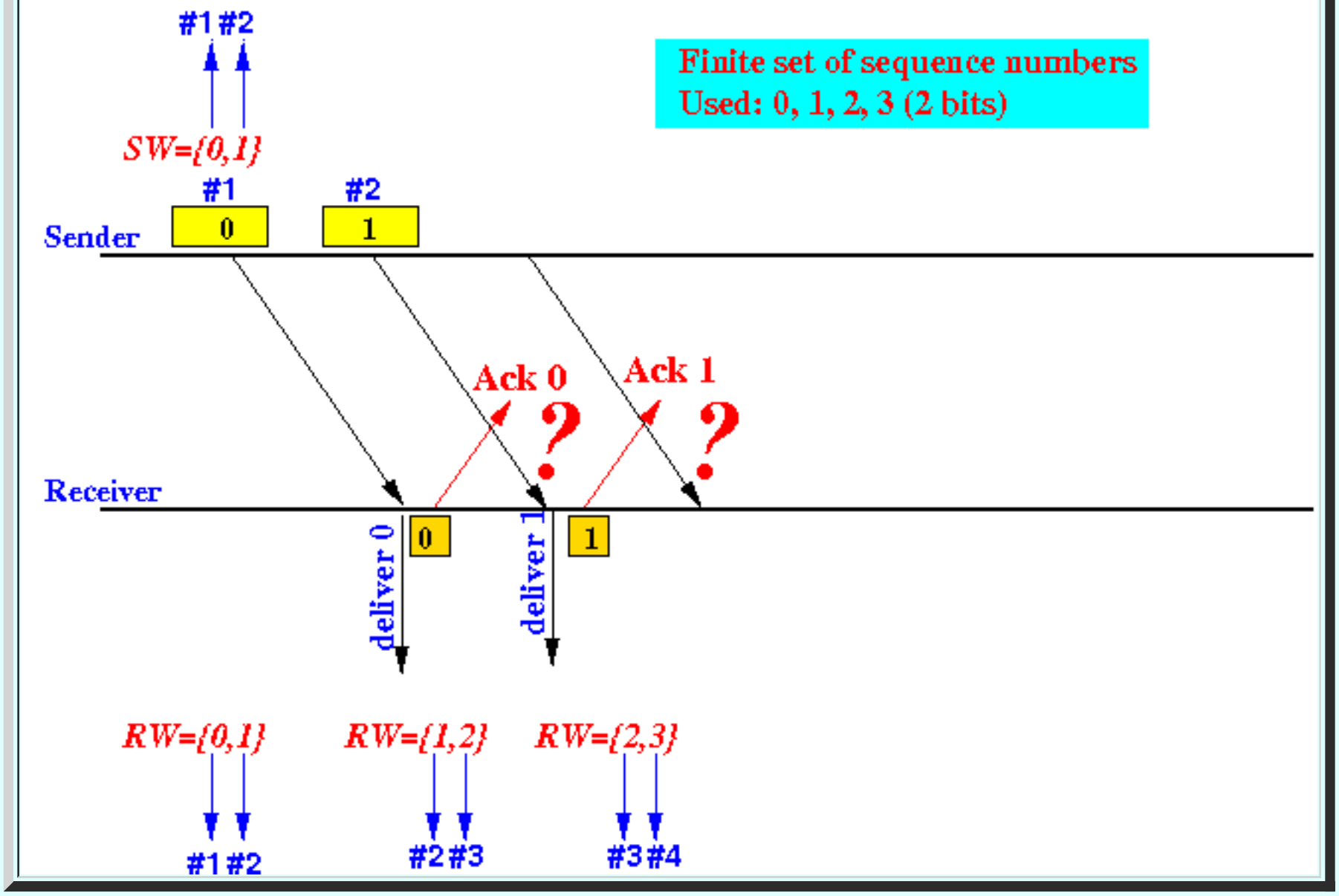
    - **Sender** sends **2 frames (#1, #2)** (due to its window size = 2) identified with **sequence numbers 0** and **1** to the **receiver**

      - **All frames** has been **received** (correctly)

    - **Receiver** sends **ACK 0** and **ACK 1**

      - The **fate** of the **ACK frames** are ***unknown***

    Graphically:



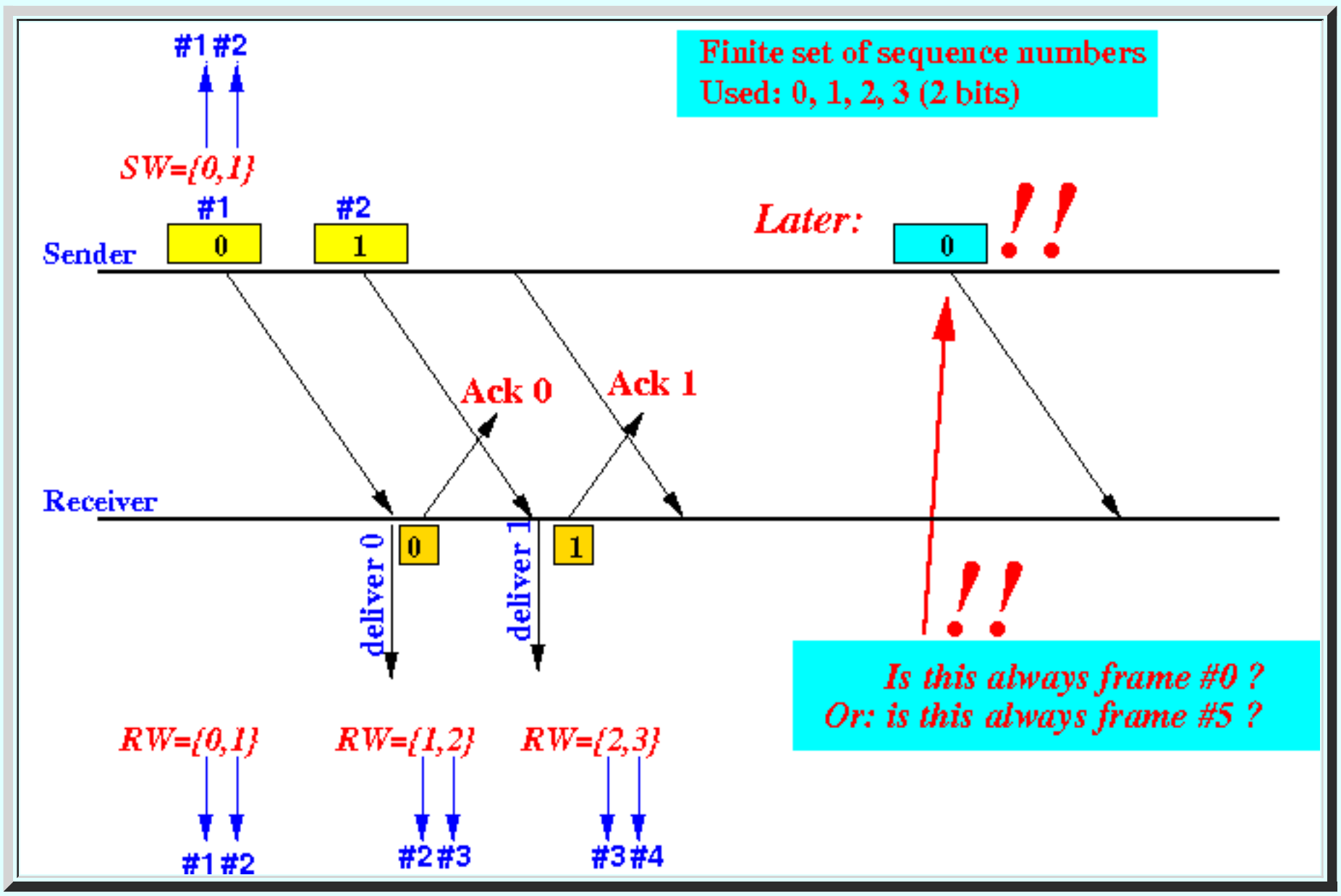
Notice that:

- **Receiver window** = **[2,3]**

  Notice:

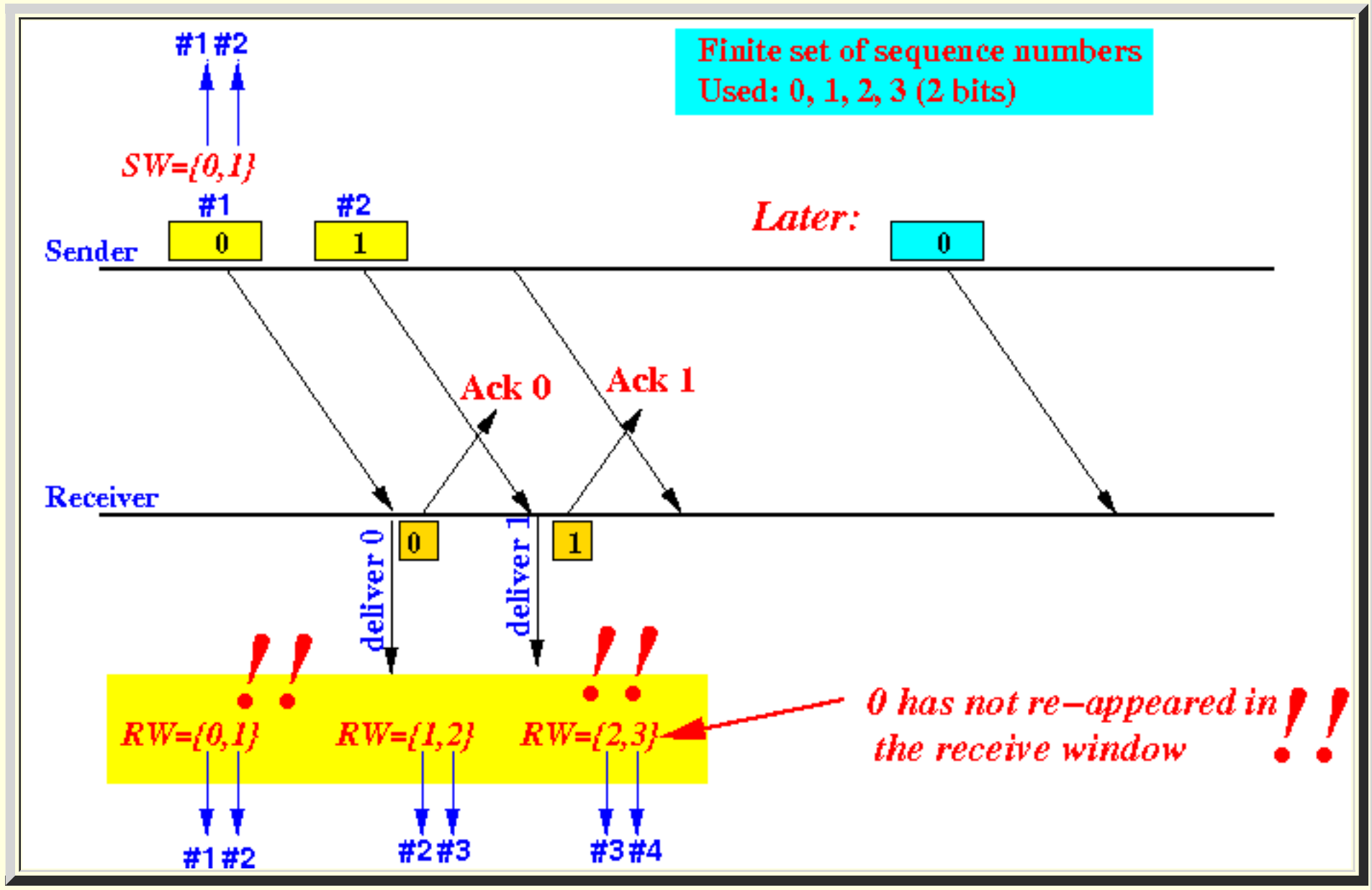
  - The **seq. no. 0** is **not** part of the **receiver window** !!!

- **Suppose** the **receiver** received a **data frame** with **sequence number = 0**:



Note:

- The **receive window** = **[2,3]**:



The **receive window** has **not wrapped around** yet !!!

- **Therefore:**

  - The **sequence number 0** (to the **receiver**) **identifies** the ***old*** **frame #1** !!!

- **$64,000 question:**

  - Is the ***second*** **frame** with **sequence number 0**:

    - ***always*** **(i.e., for *sure*)** the **frame #1** from the **sender ??????**

    (Or **can** it be **frame #5** under ***some*** **circumstances** ????)

  - **Answer** to the **$64,000 question**:

    - The ***second*** **frame** with **sequence number 0** is ***always*** the **frame #1** !!!

- **Reason why the second frame with sequence number 0 is always the frame #1**

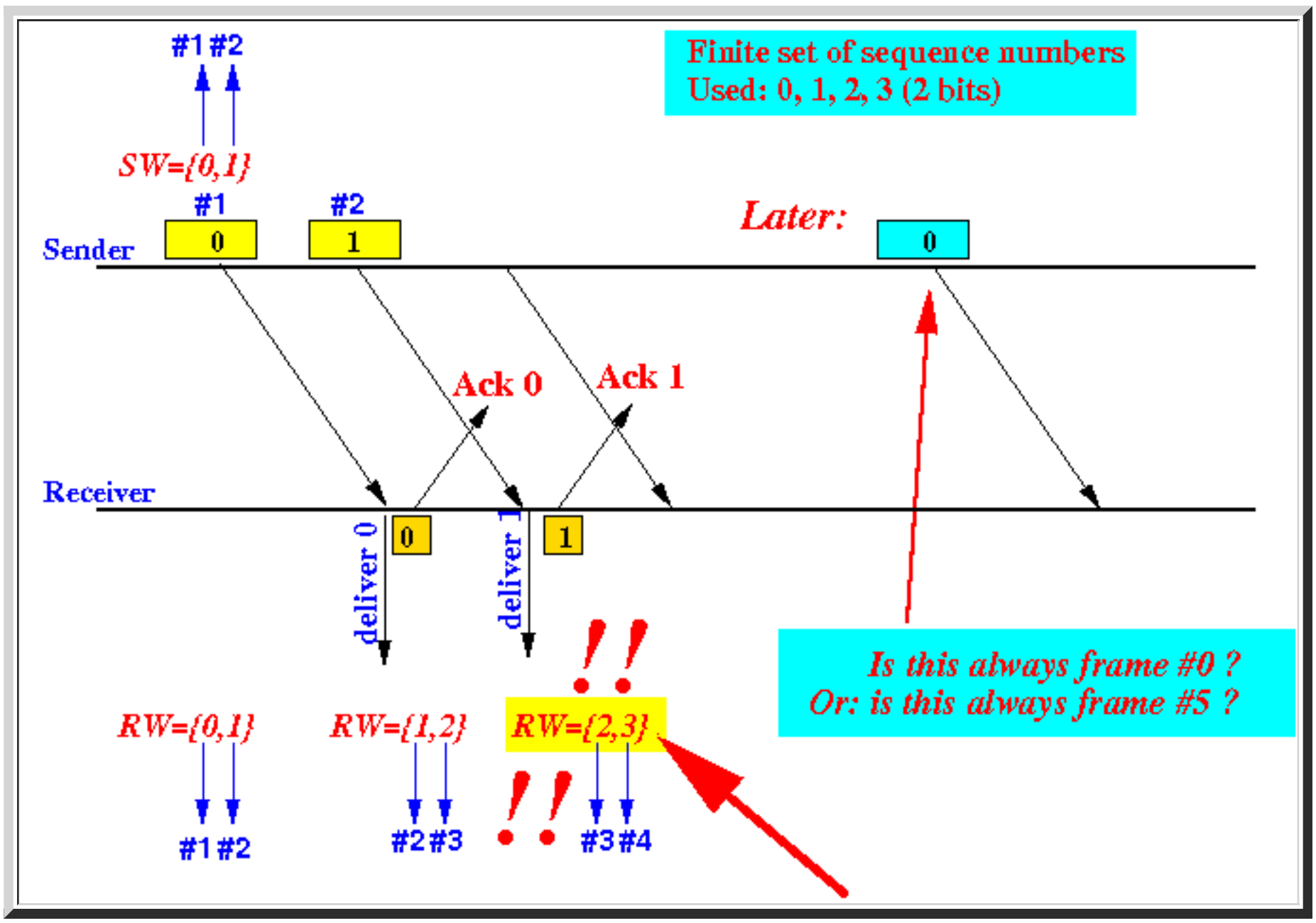
  - **Recall** that:

- The **receive window** is ***always*** **ahead** of the **send window**

I.e.:

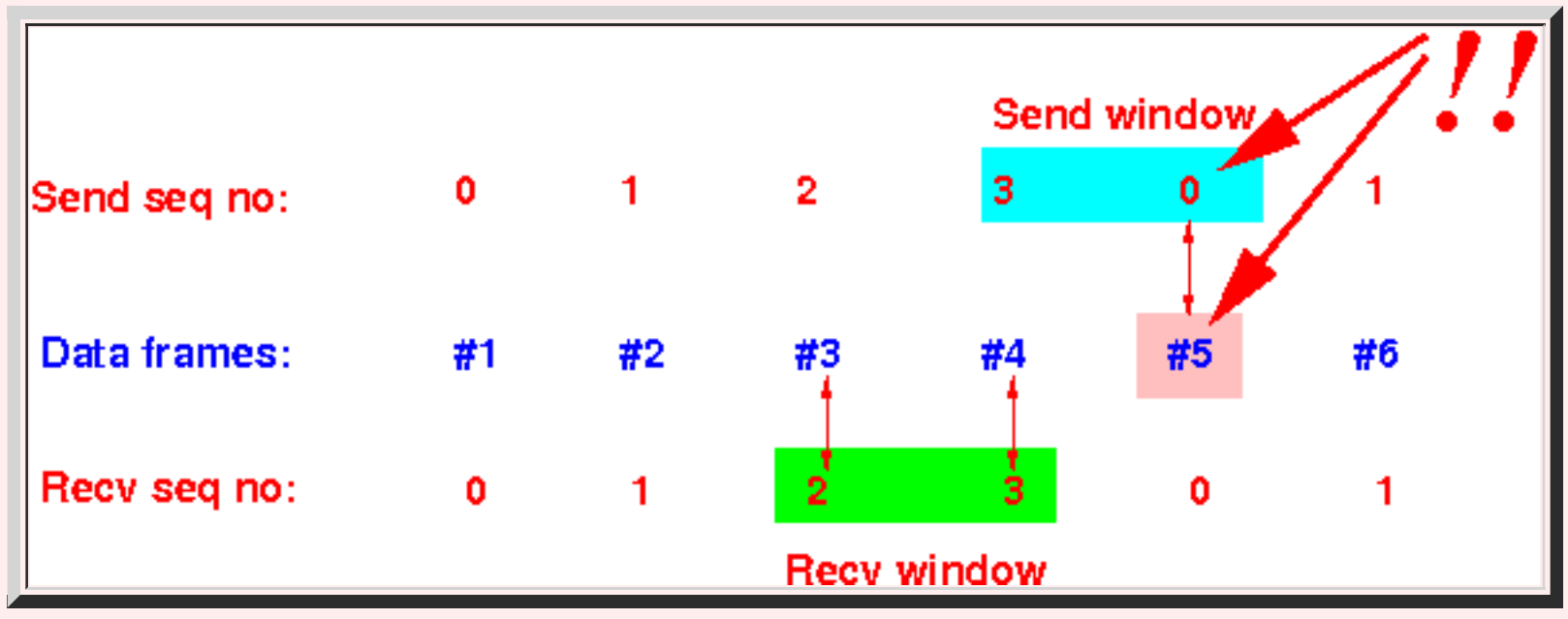
- The **send window** can ***never*** get **ahead** of the ***receive*** **window** !!!

- The **receive window** is **[2,3]**:



- ***Therefore:***

- ***If*** the **sender** **is using** the **sequence number 0** to **identify** the **frame #5**, then we have **this situation**:

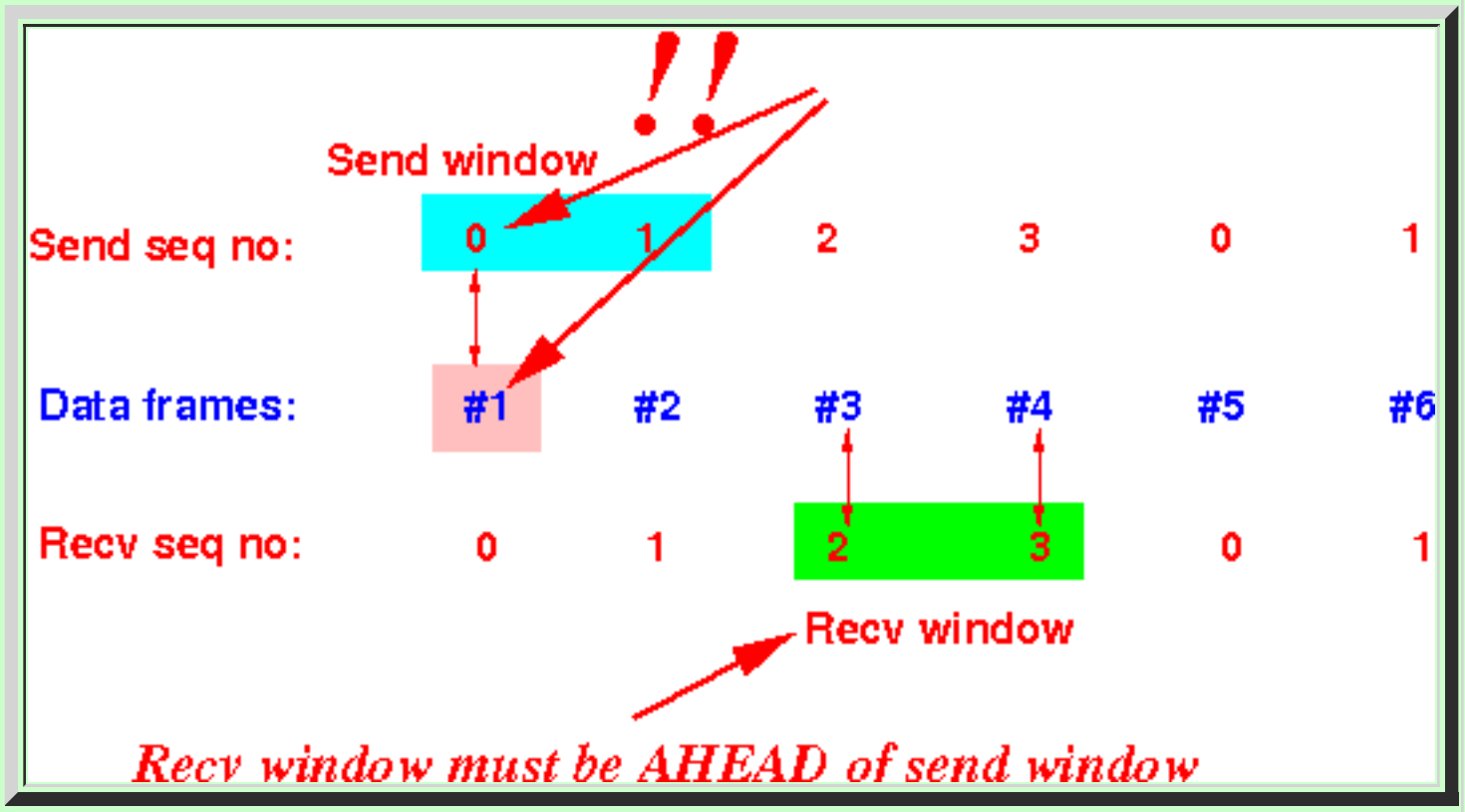


This situation is ***not*** **possible** because:

- The **send window** is ***ahead*** of the **receive window** !!!

- **Conclusion:**

- The ***only*** possible **frame** that can be **identified** by **seq no 0** is **frame #1**:



There is ***no*** **ambiguity possible** !!!!

Condition for unambiguity with finite sequence numbers

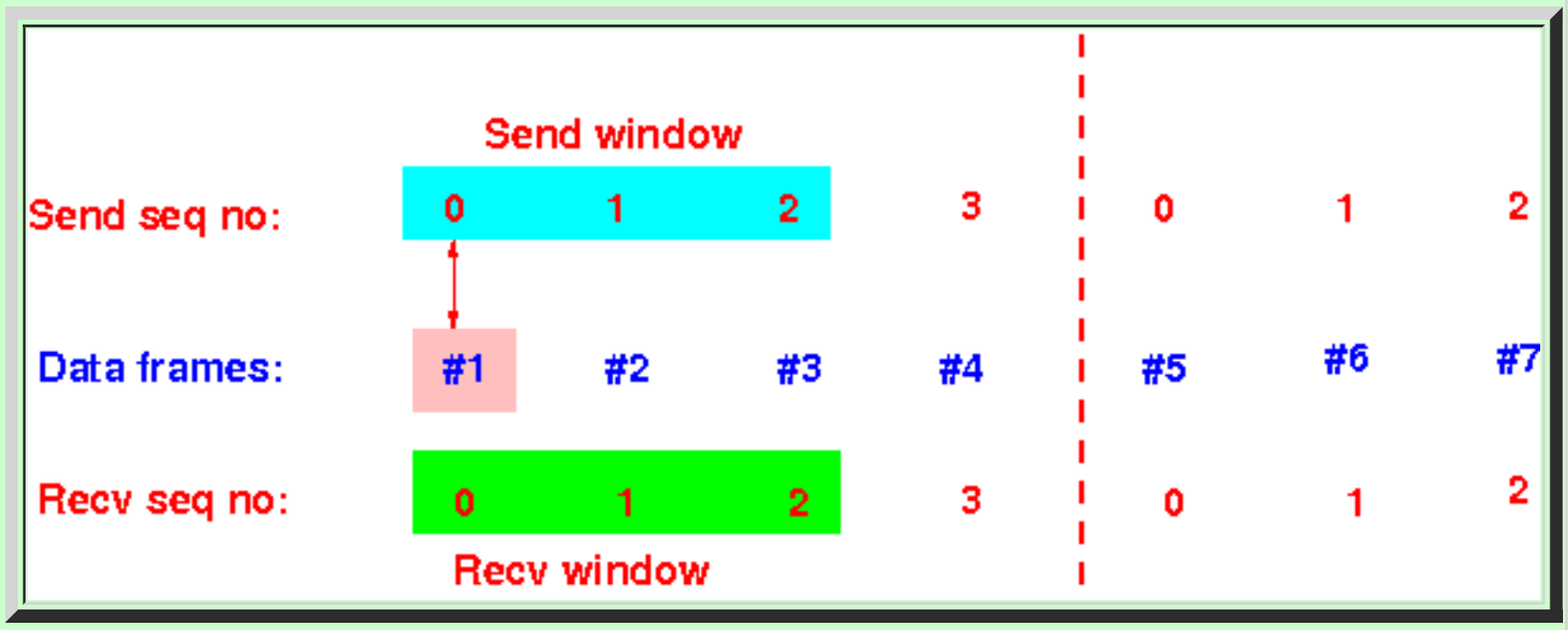
- **Pre-requisite to understanding this material: relationship between the send window and receive window**
  - **Consider** the **following** **sliding window setting**:

    - **N = 4** (we use **4 sequence numbers**)

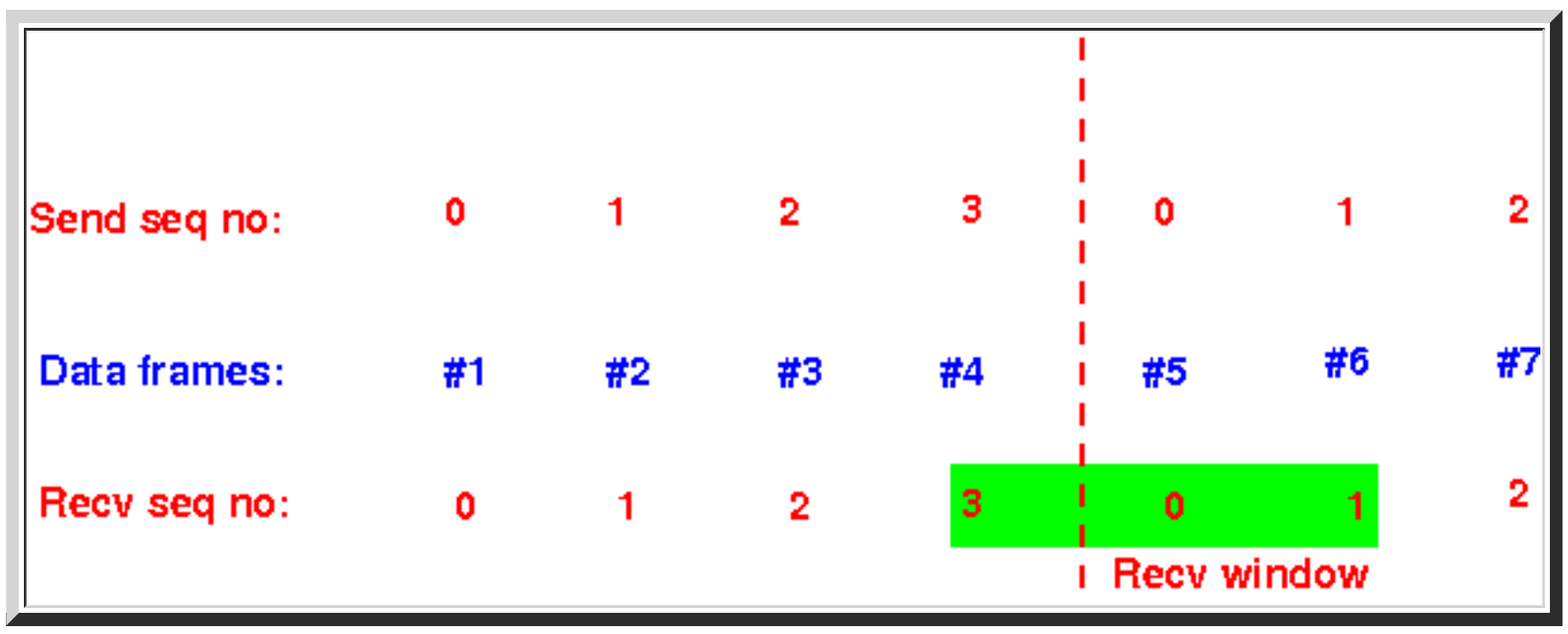
    - **Send window size** = **3**
    - **Receive window size** = **3**

  - **Initially**:

    - The **send/receive windows** are **lined up**



  - **Suppose** the **receive window** is now: **[3,0,1]**



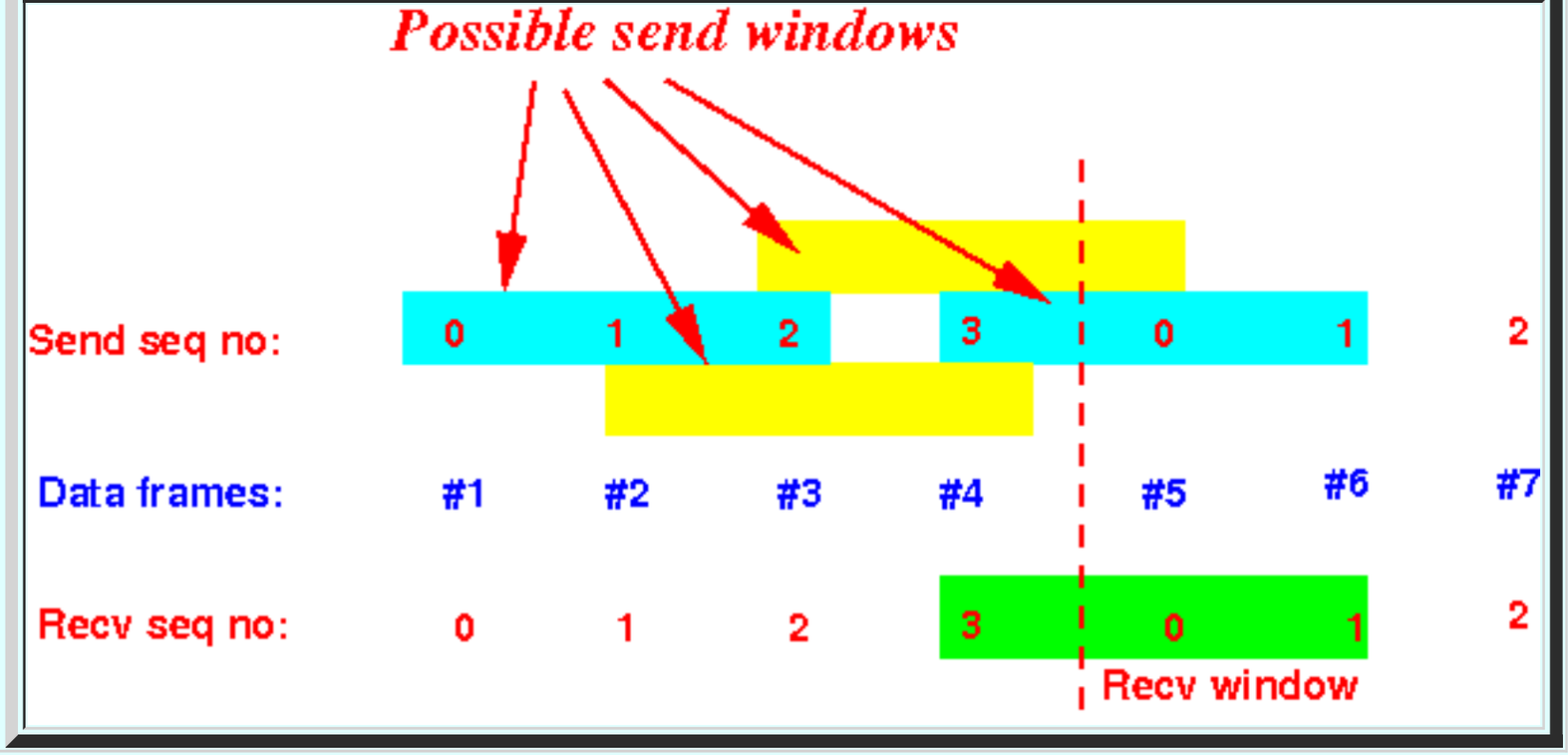
Question:

- **What** are the **possible values** for the **send window** ???

Answer:

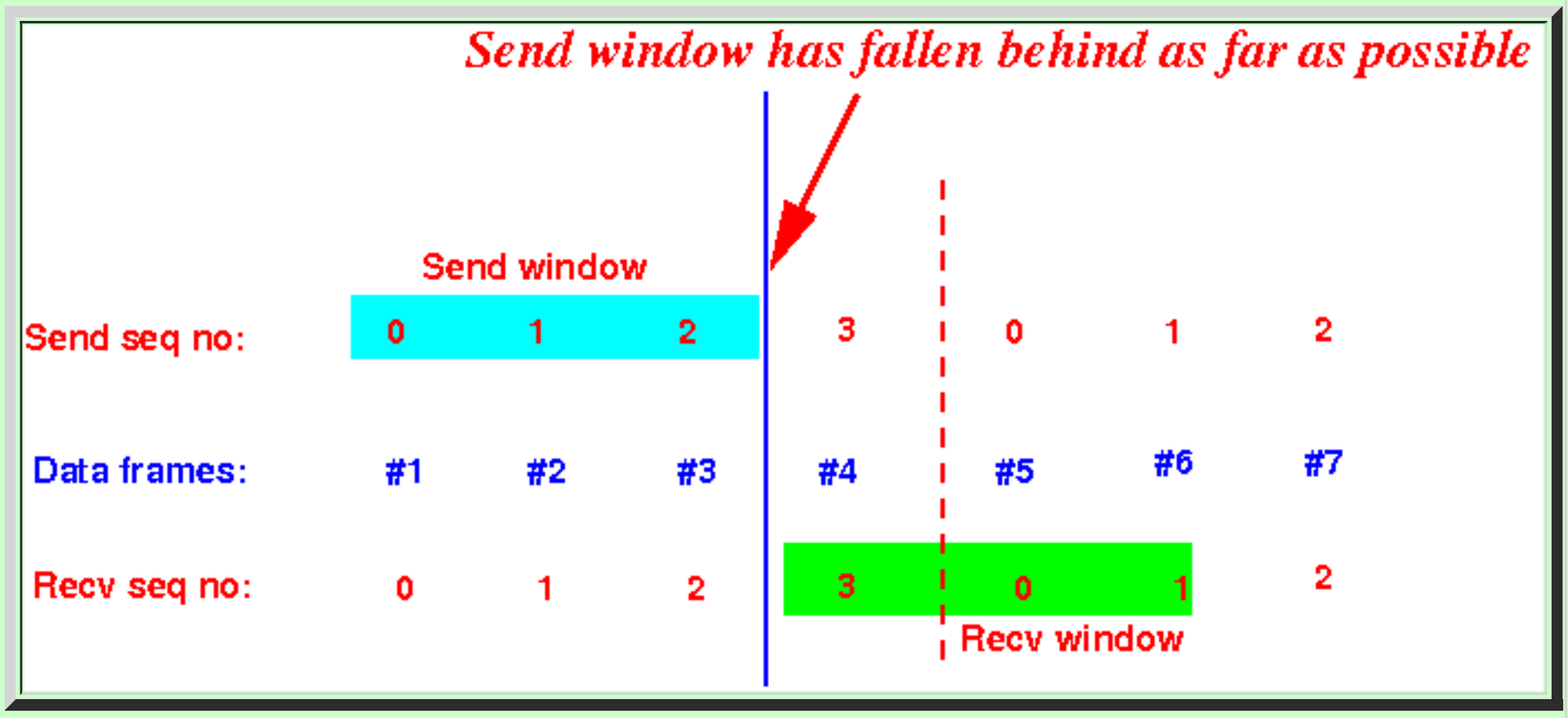
- The **possible values** for the **send window** are:
  - **[0,1,2]**
  - **[1,2,3]**
  - **[2,3,0]**
  - **[3,0,1]**

Graphically:



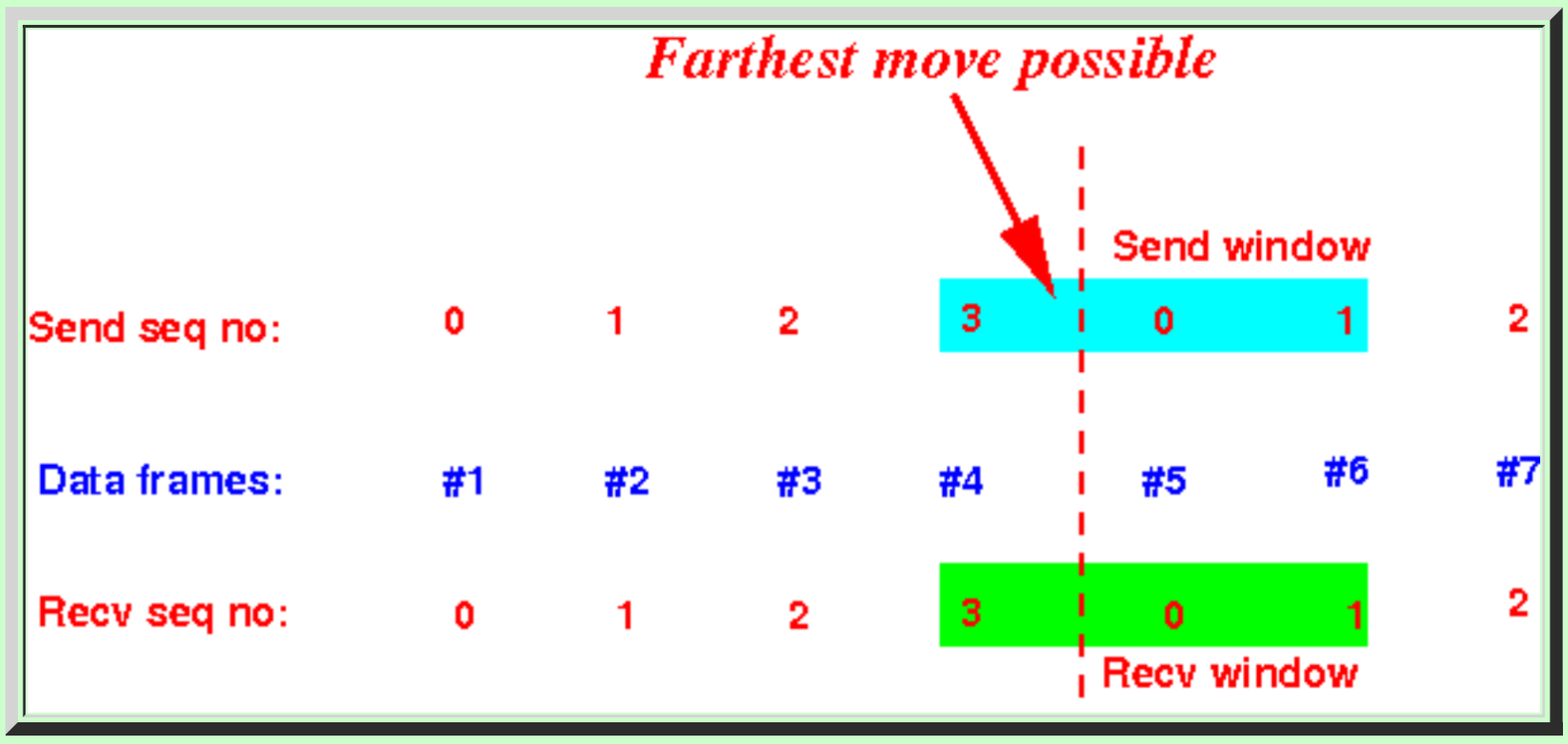
Reason:

- The **send window** can fall the ***farthest*** **behind** the **receive window** as **possible**:



This **happens** when ***all*** **ACK frames** were **lost**

- The **send window** can ***line up*** **(= farthest ahead)** with the **receive window**:



This **happens** when ***all*** **ACK frames** were ***received***

- When **some** of the **ACK frames** were **received** (and some were lost), the **send window** can be **[1,2,3]** or **[2,3,0]**

- **How is ambiguity "created"**

  - **Fact:**

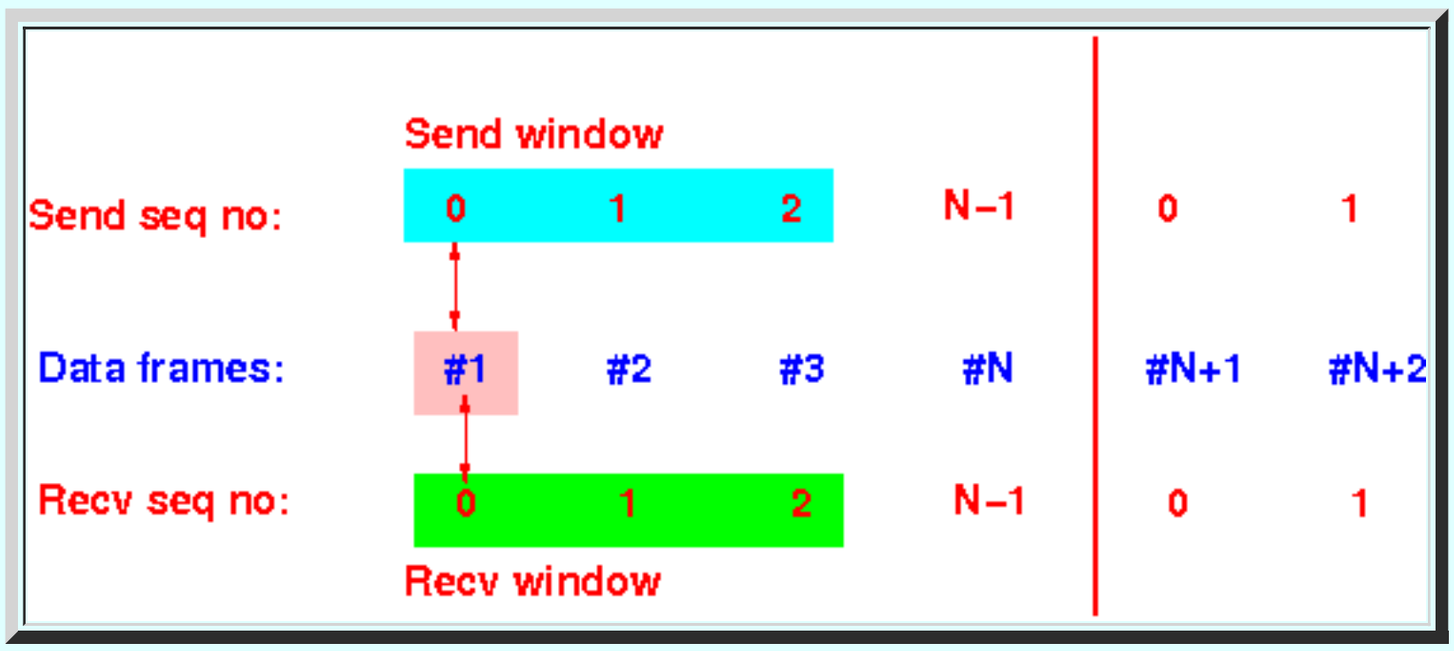
    - **Ambiguity** is **"created" (= caused)** by a **sequence number *x*** when:

      1. The **sequence number *x*** is **inside** the **receiver window**

      2. The ***sender*** can **use** the **sequence number *x*** to **identify** ***two (2)*** **different (data) frames**

Example:

- **Initially**, the **send window** and **receive window** are **lined up**:



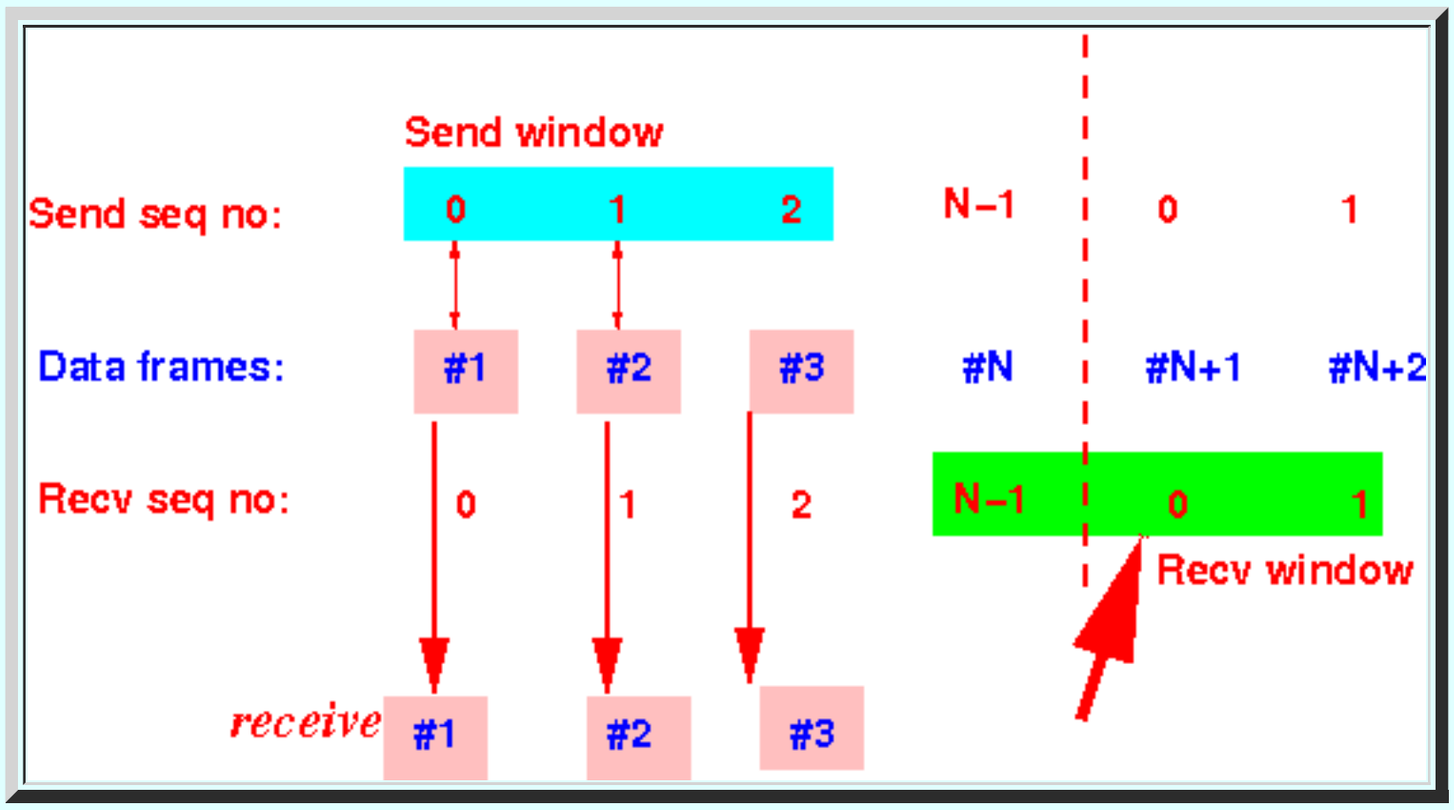
Without loss of generality, we can **assume** that:

- The **sender window** and **receiver window** are **lined up** at the **sequence number 0**

  (The argument will work for ***any*** **sequence number**.

  It is **easier** to understand using as ***starting*** **sequence number** = **0**)

- **Suppose** the **receiver** receives **3 frames** and **slides** the **receive window** forward:

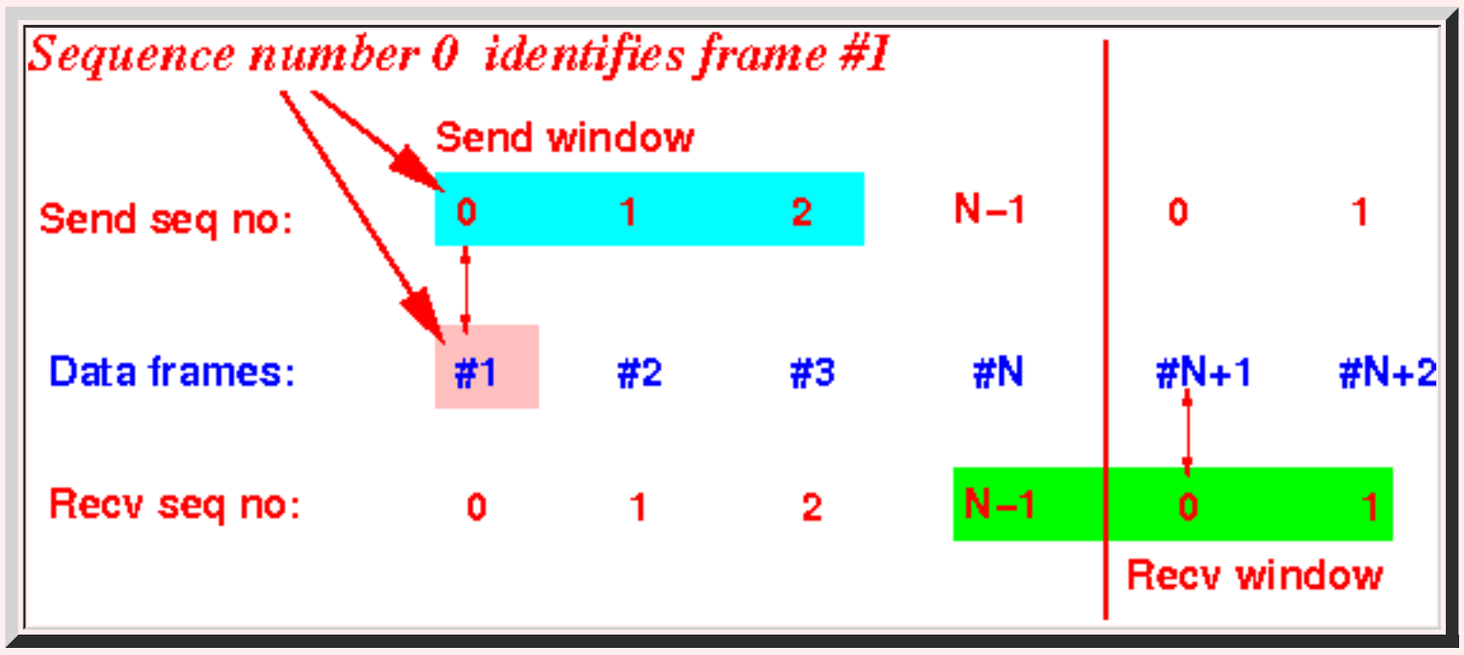


- **Pay attention** the **sequence number 0** inside the **receive window**

  It is **now** ***possible*** for the **sender** to use ***confuse*** the **receiver** by using **sequence number 0**:

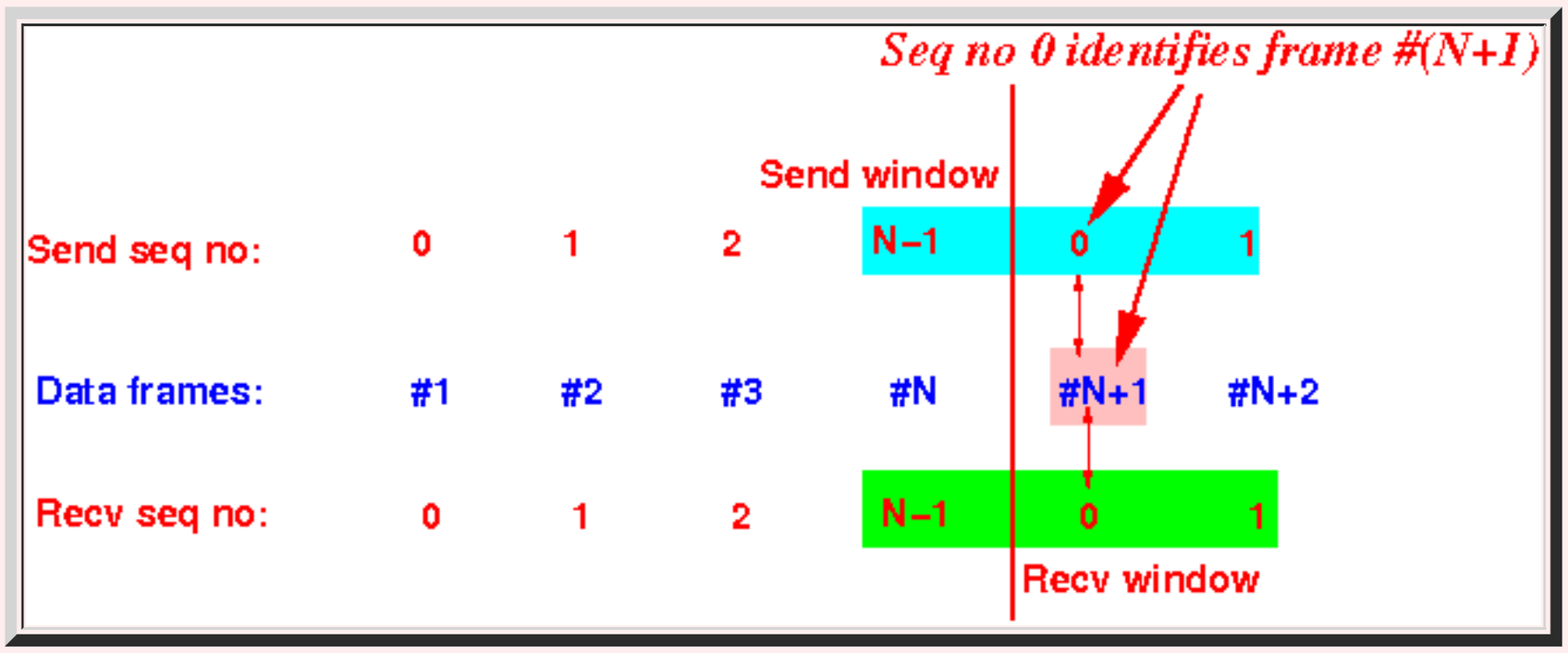
  1. The **sender could** have **fallen behind** and **re-transmit** the **frame #1** using the **sequence number 0** to the

receiver:



---

2. The **sender could** have **caught up** and **sends** the **frame #(N+1)** using the **sequence number 0** to the **receiver**:



- **How to prevent ambiguity**:

  - For **every sequence number *x*** inside the (= any) **receive window**

    - We **make sure** that the **sender cannot** use **sequence number *x*** to **identify *two* different frames** !!!!!!

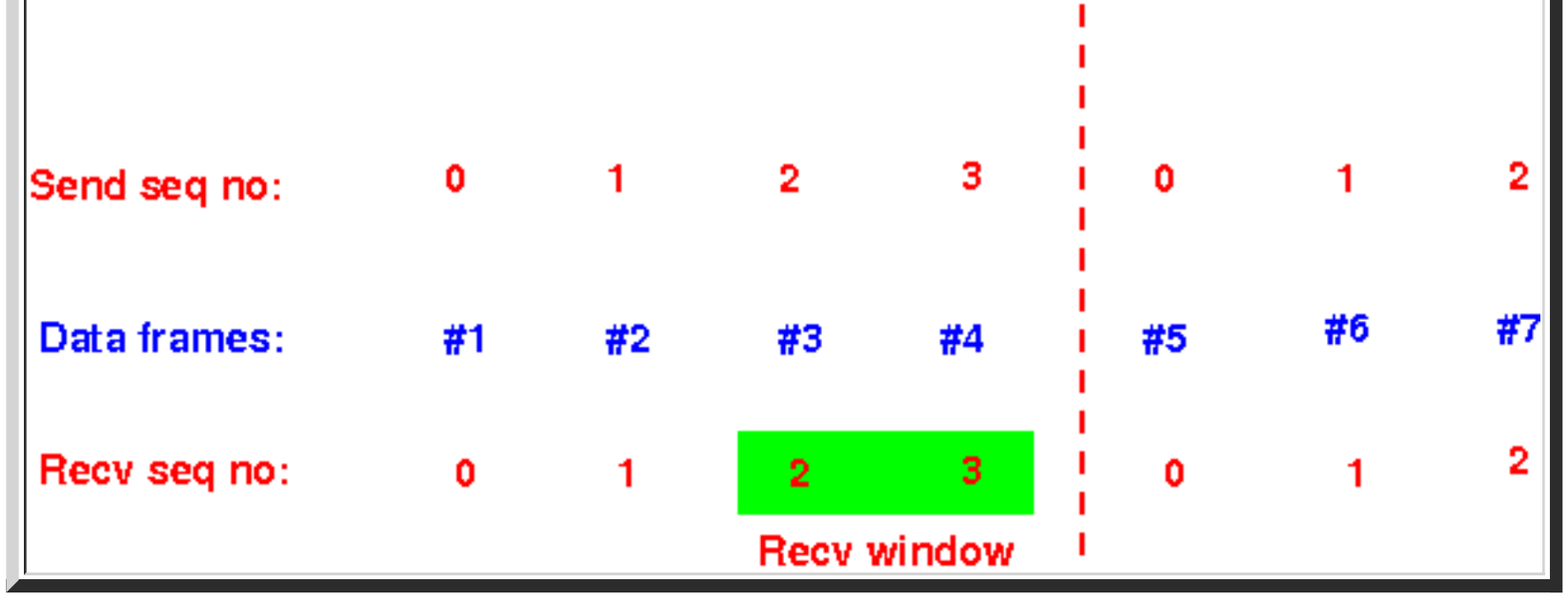
- **A concrete example on how to prevent ambiguity**

  - **Consider** the **following sliding window setting**:

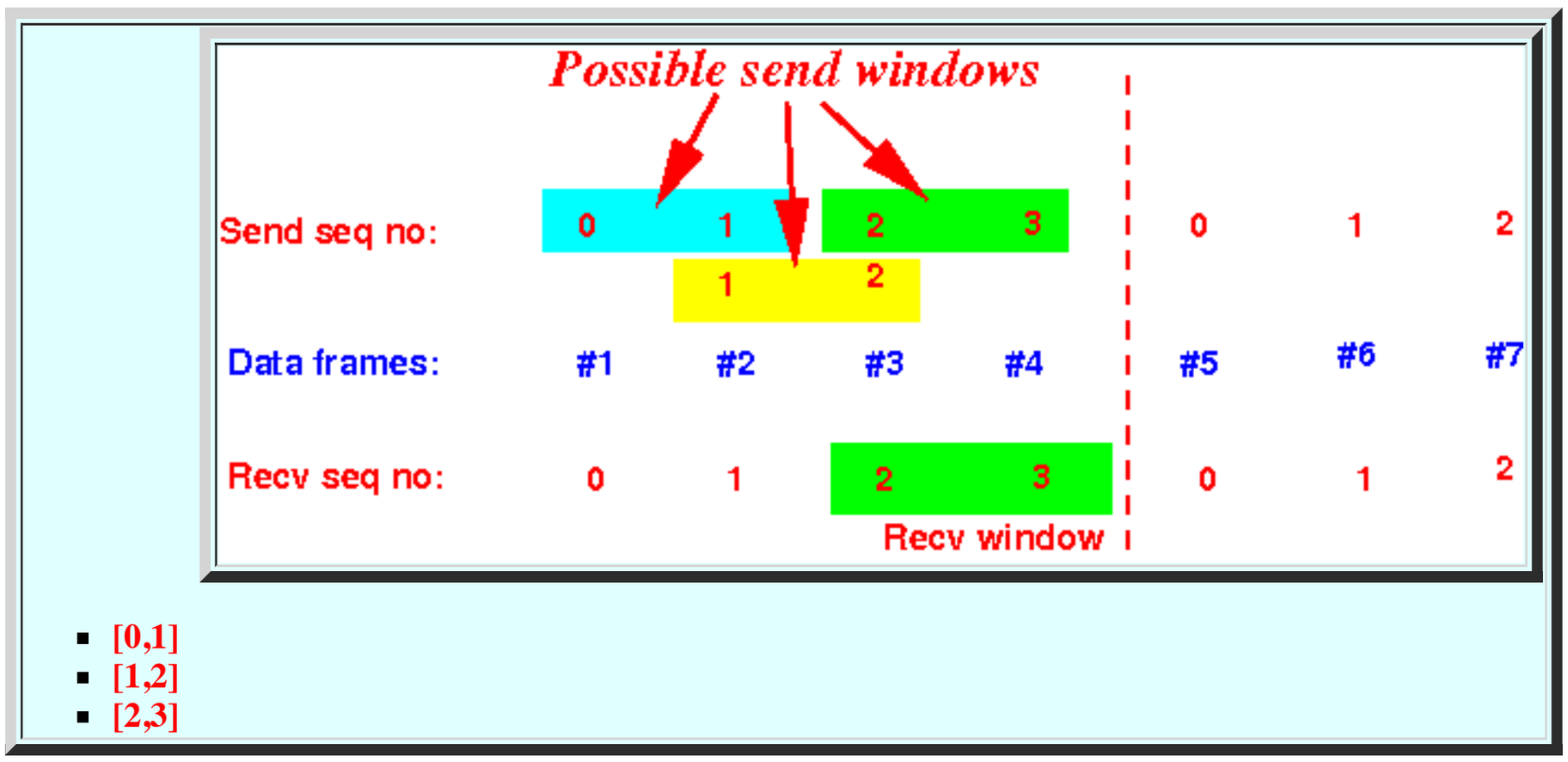
    - **N = 4** (we use **4 sequence numbers**)

    - **Send window size** = **2**
    - **Receive window size** = **2**

  - **Suppose** that the **receive window** is **[2,3]**:



- The **possible values** of the **send window** of the **sender** are

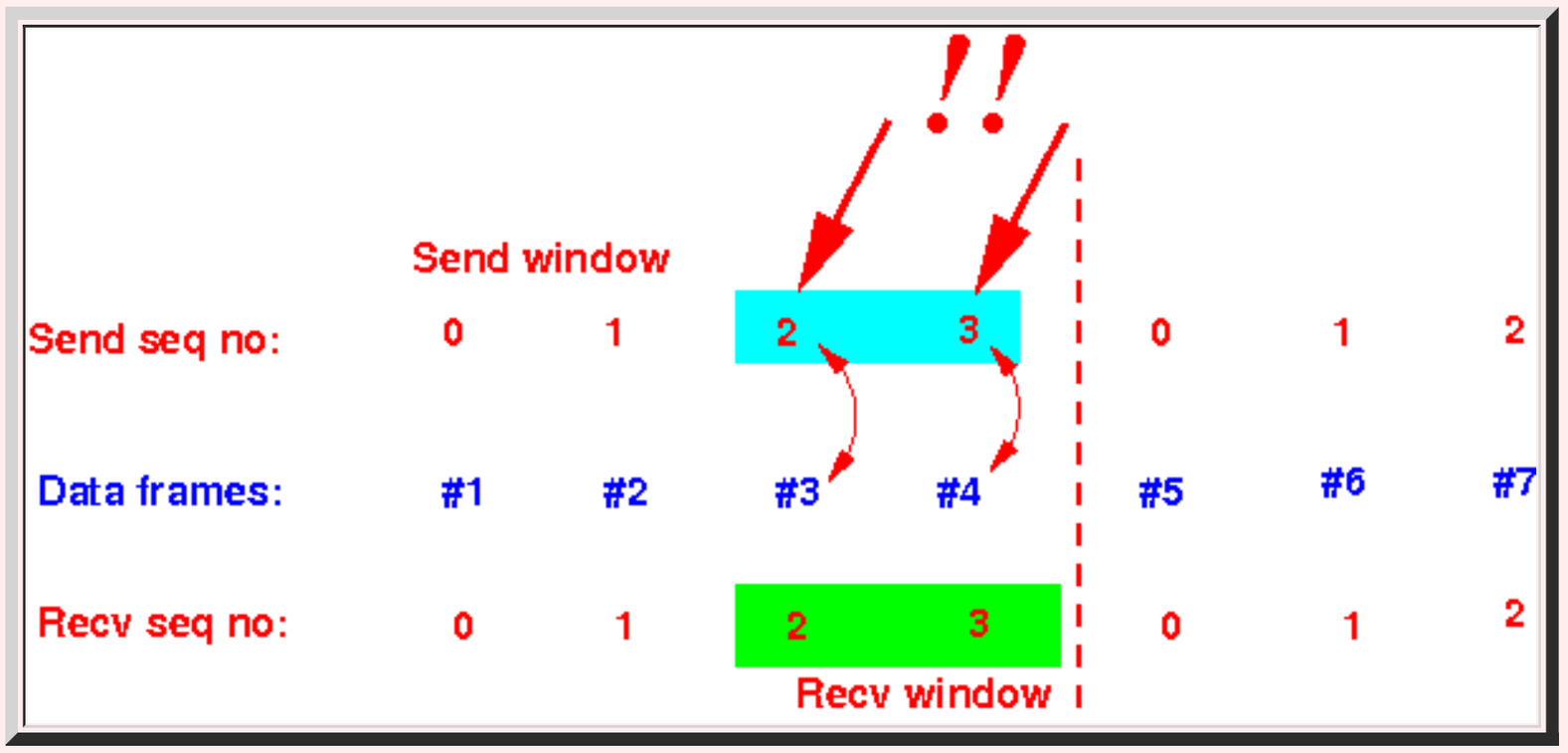


  - **[0,1]**
  - **[1,2]**
  - **[2,3]**

- The **sender cannot** confuse the **receiver** because:

1. The **sender** can ***only*** use the **sequence number 2** to **identify** **frame #3**

   The **sender** can ***only*** use the **sequence number 3** to **identify** **frame #4**



The **sender cannot** use the **sequence number 2 or 3** (in the **receive window**) to **identify** ***2 different*** **frames** !!!

Therefore:

- **No ambiguity possible** !!!!!!

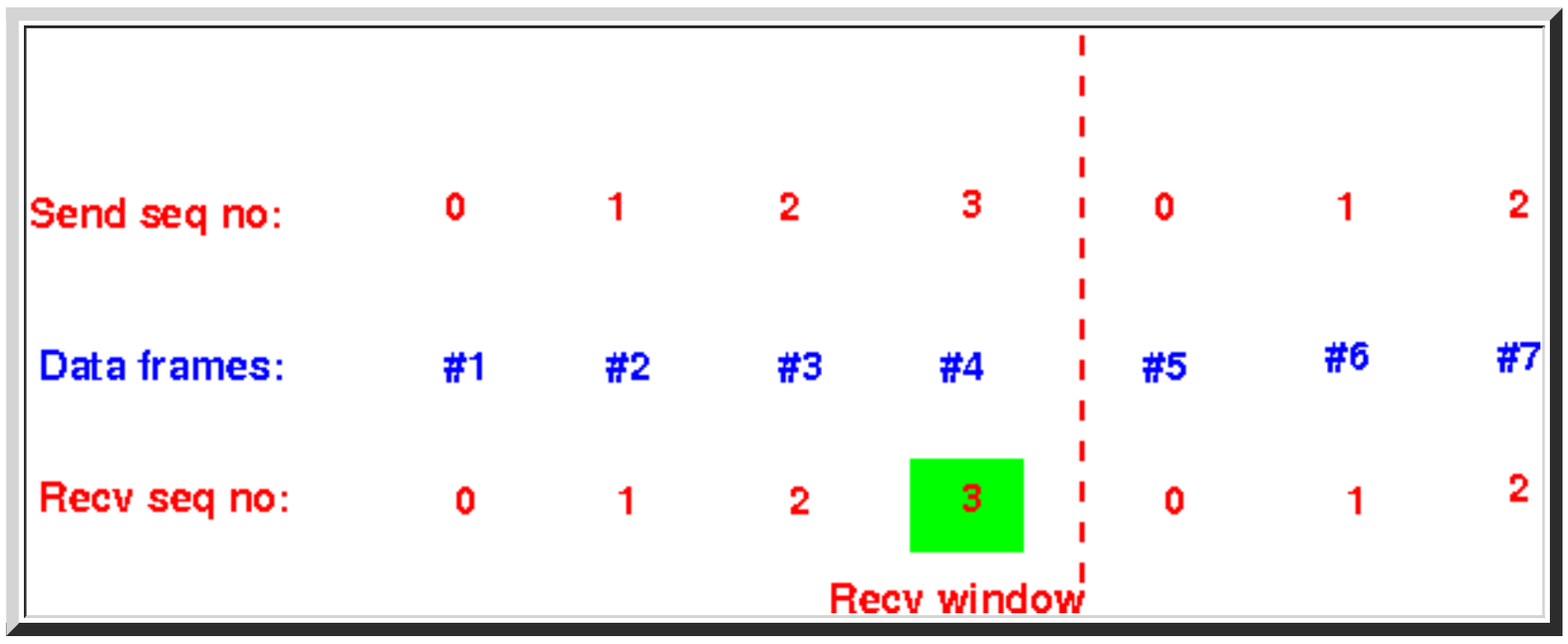
This *unambiguous case* was **discussed** **previously**: **click here**

- **Another example to how to prevent ambiguity**
  - **Consider** the **following** **sliding window setting**:

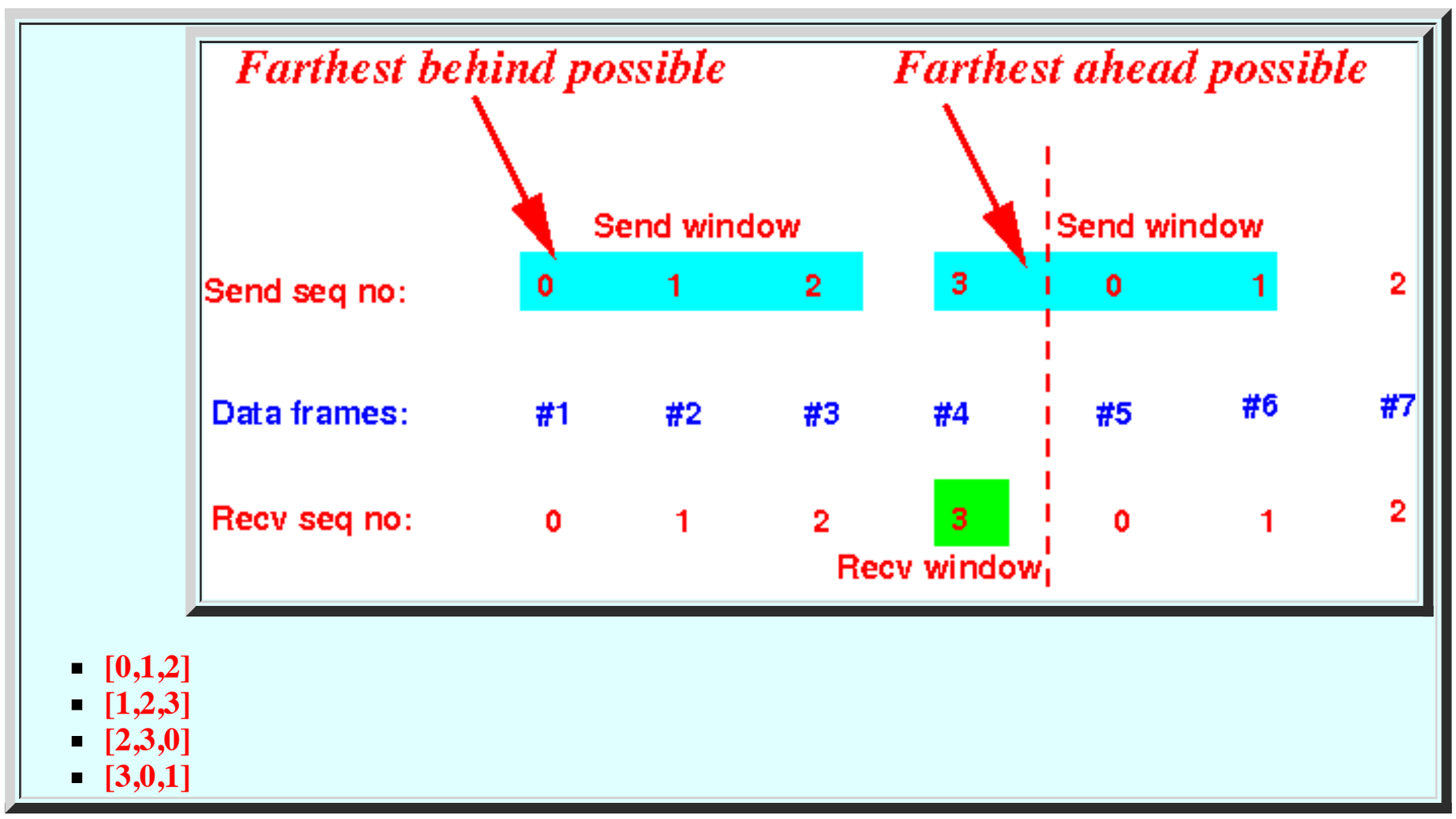
    - **N = 4** (we use **4 sequence numbers**)

    - **Send window size** = **3**
    - **Receive window size** = **1**

  - **Suppose** that the **receive window** is **[3]**:



  - The **possible values** of the **send window** of the **sender** are:



    - **[0,1,2]**
    - **[1,2,3]**
    - **[2,3,0]**
    - **[3,0,1]**

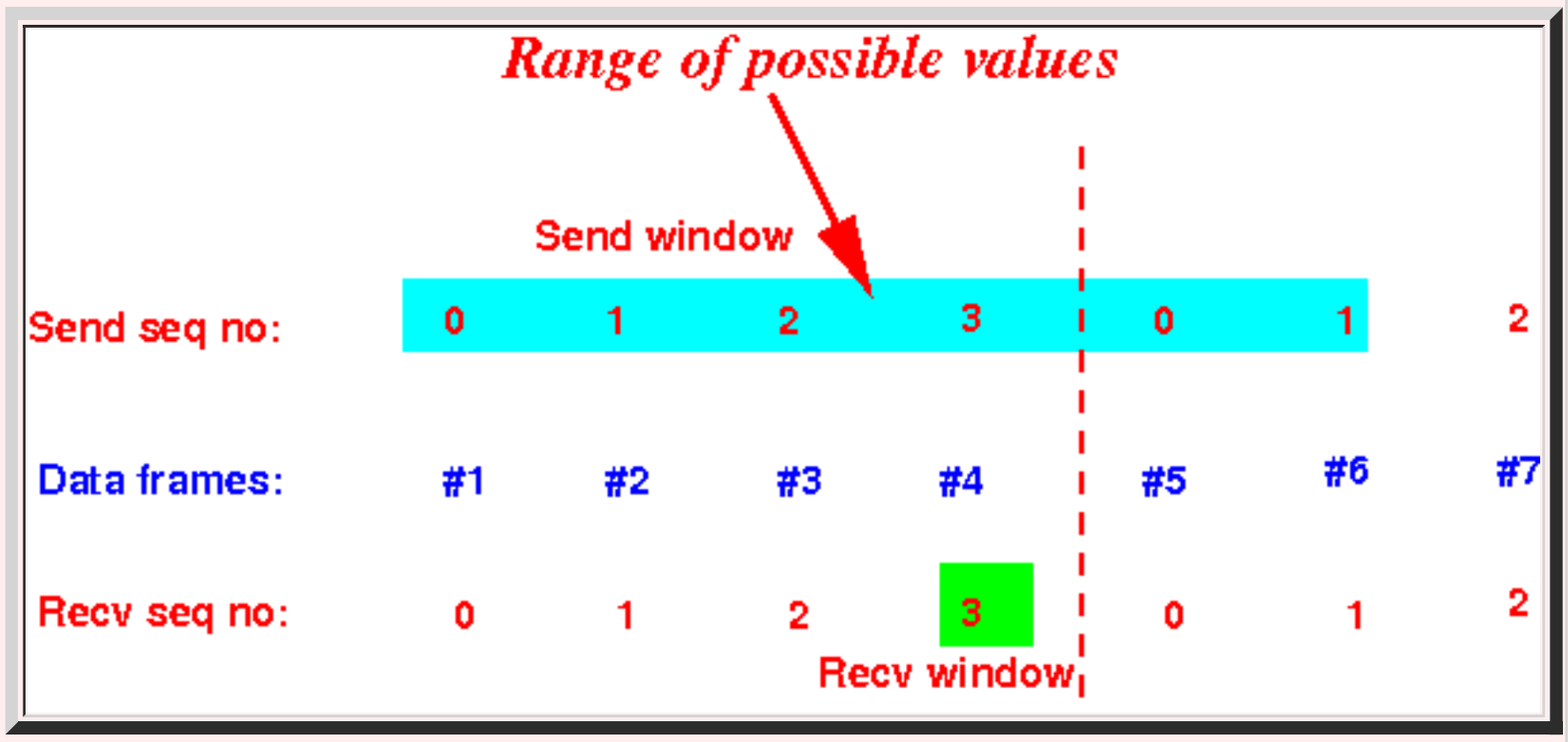
  - The **sender cannot** confuse the **receiver** because:

    1. The **receive window** = **[3]**

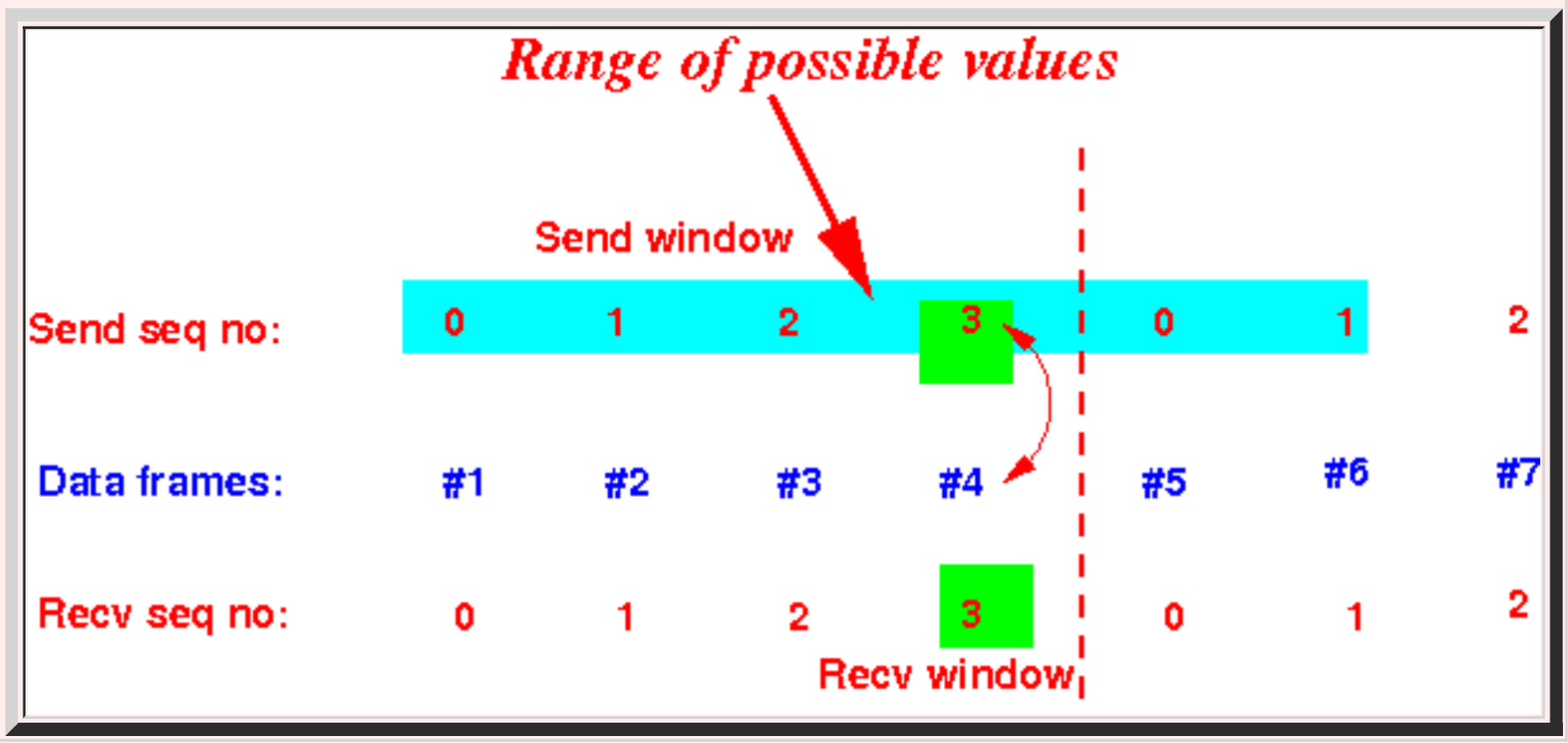
       That mean:

       - The **receiver** will ***only*** **buffer** the **frame** with **sequence number 3** !!!

2. Within the **range** of the **send window values**:



The **sender** can ***only*** **transmit** the **frame #4** using the **sequence number 3**:



- **Sufficient condition to guarantee unambiguity (with proof)**
  - **Sufficient condition** that **guarantee** that a **sequence number** in the **send window** will ***uniquely*** identifies a **unique frame**:

    - Let N = the **number** of **different sequence numbers** available
    - Let W_{send} = the **send window *size***
    - Let W_{recv} = the **receive window *size***

    - ***If:***

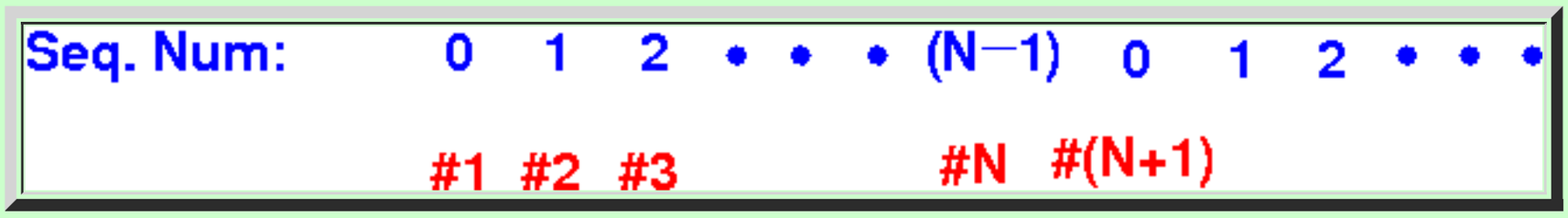
      (1) $W_{send} + W_{recv} \leq N$

      then:

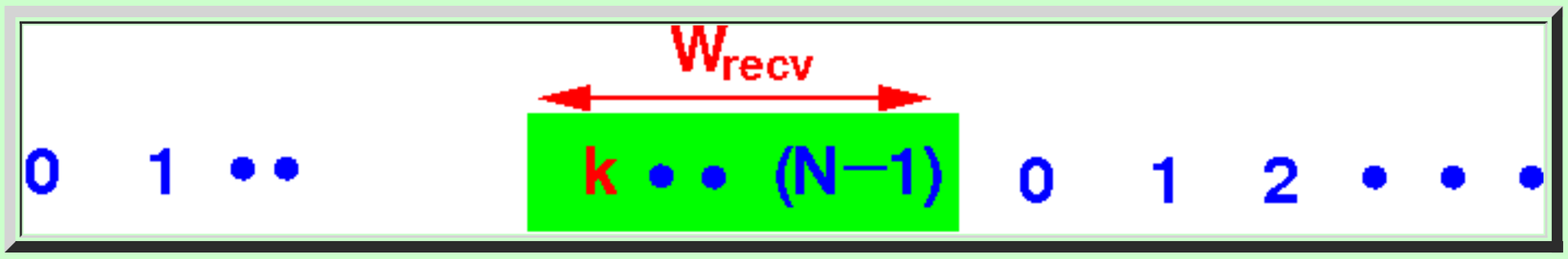
      - A **sequence number** in the **send window** will ***uniquely*** identifies **one specific frame**

Proof:

- The **labeling** of the **data frames** is as **follows**:

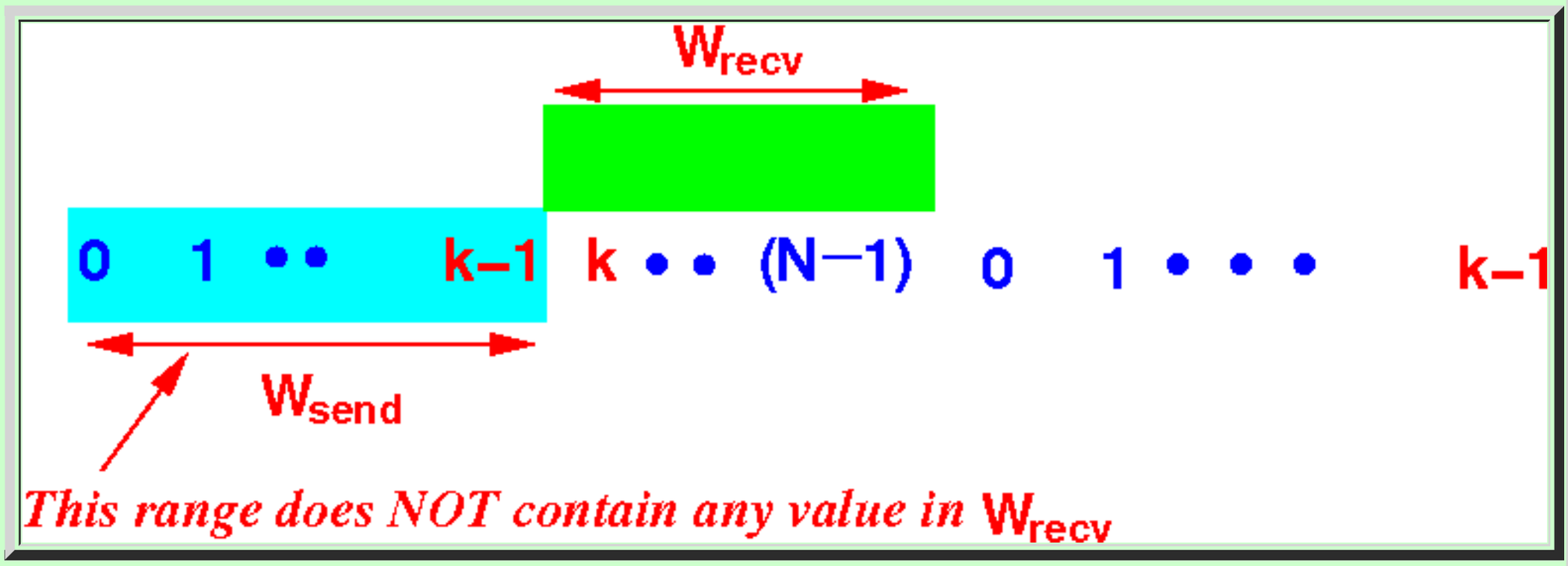


- **Consider** an **arbitrary receive window**:

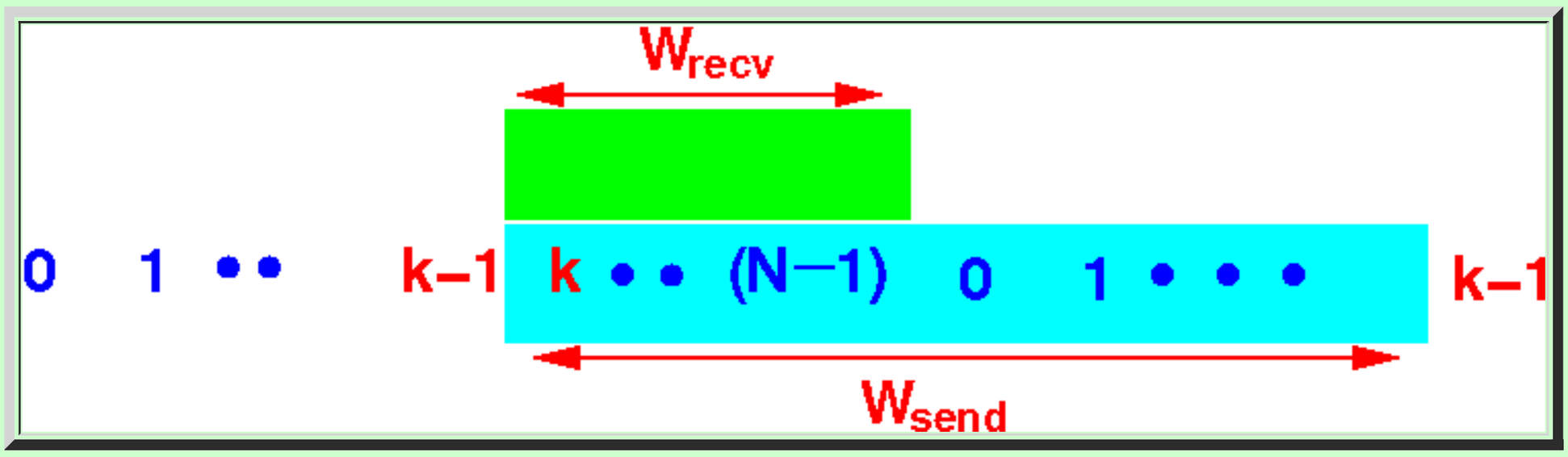


(The **proof** is **similar** for **other windows**, but **this specific window** make the **explanation easier** to **understand**)

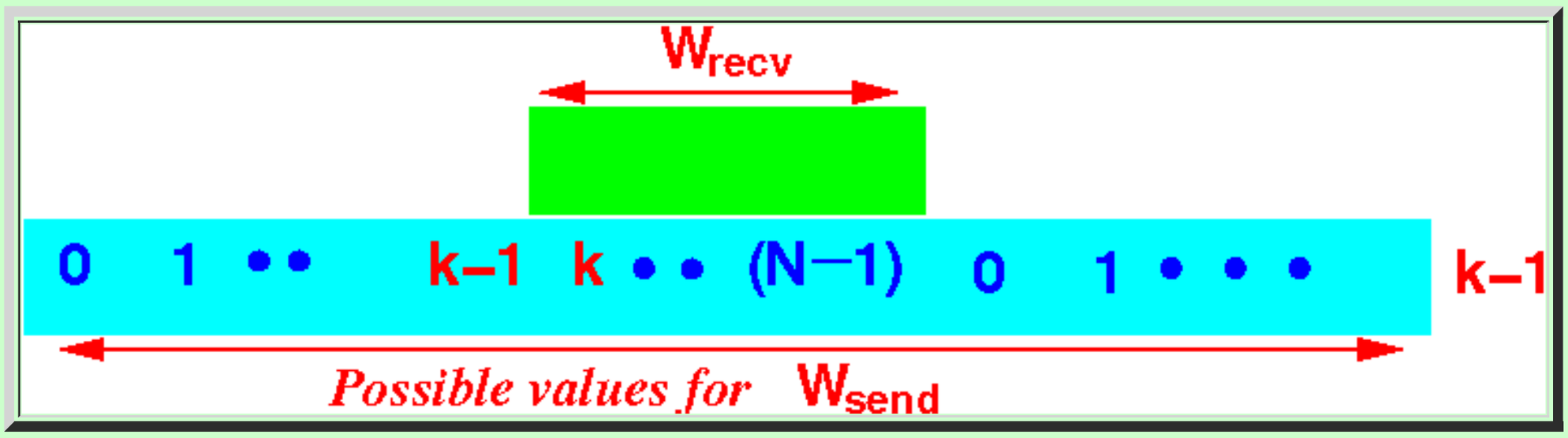
- The **farthest *behind* value** for the **send window** is:



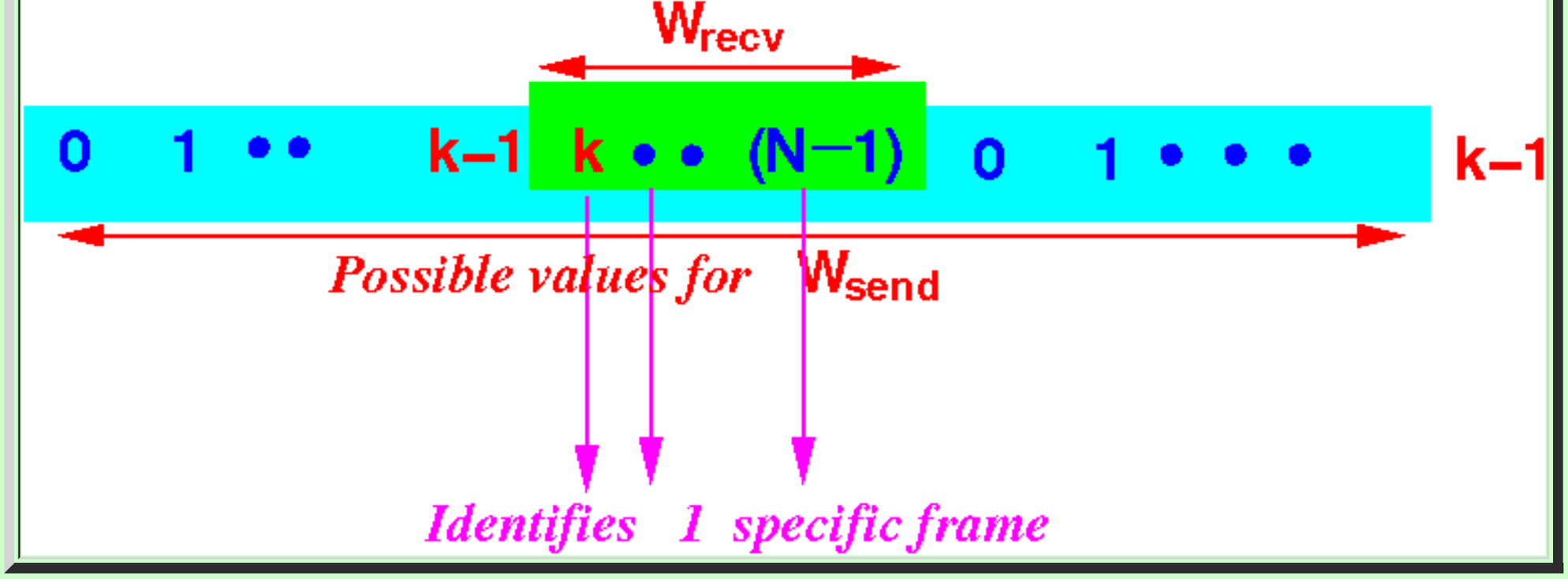
- The **farthest *ahead* value** for the **send window** is:



- **Within** the ***range*** of **possible values** for the **send window**:



each value **inside** the **receive window** (W_{recv}) is used to **identify** ***one*** **(1) specific frame**:



Therefore:

- **Ambiguity** is ***not*** **possible**

- **Commonly used Send/Receive windows**
  - If the **frame** has ***N*** **bits** for **sequence/ACK numbers**, then:

$$W_{send} = 2^{N-1}$$
$$W_{recv} = 2^{N-1}$$

Go-Back-N: sliding window algorithm using receive window size = 1

- **Go-Back-N**
  - **Go-Back-N**:

    - **Go-Back-N** = a **special representative** in the **sliding window** **protocol family** that uses:

      - $W_{send} = N-1$
      - $W_{recv} = 1$

- **Stop-and-Wait**
  - **Stop-and-Wait**:

    - **Stop-and-Wait** = another **special representative** in the **sliding window** **protocol family** that uses:

      - $W_{send} = 1$
      - $W_{recv} = 1$