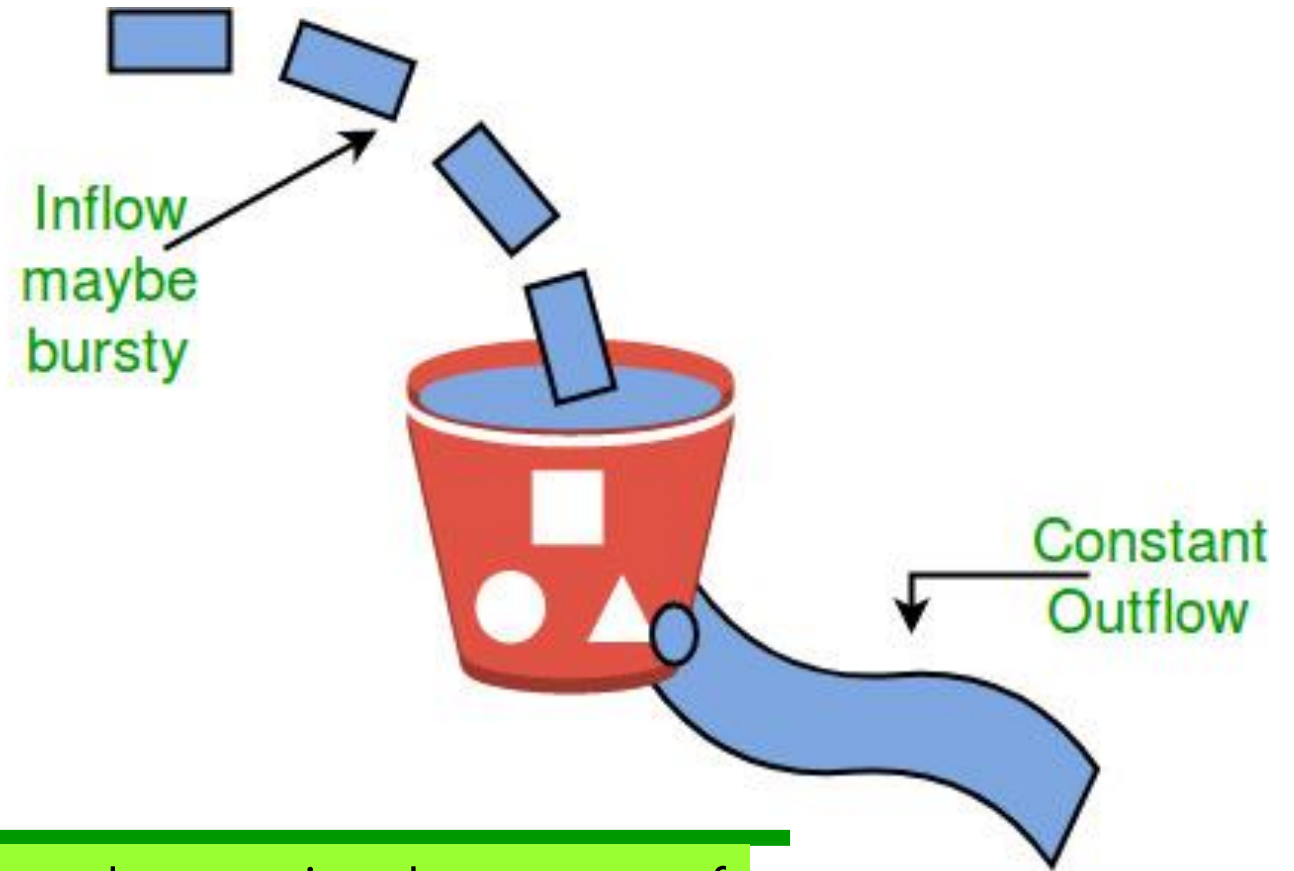


# Data link Layer Flow Control

# Flow Control

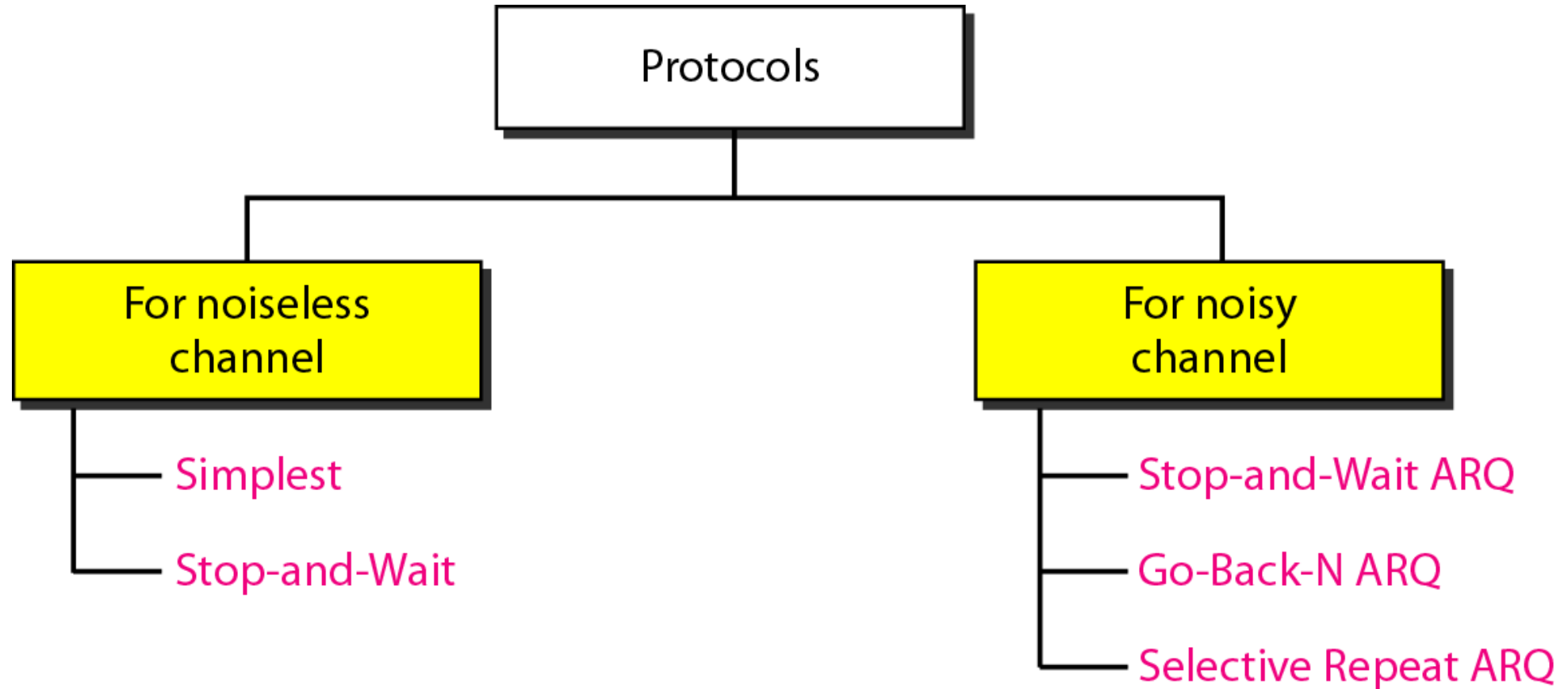
- Flow Control coordinates the amount of data that can be sent before receiving an acknowledgement.
- The flow of data must not be allowed to overwhelm the receiver.
- Any receiving device has a limited speed at which it can process incoming data and a limited amount of memory in which to store incoming data.
- The rate of such processing is often slower than the rate of transmission. For this reason, each receiving device has a block of memory, called a buffer, reserved for storing incoming data until they are processed.
- If the buffer begins to fill up, the receiver must be able to tell the sender to halt transmission until it is once again able to receive.

# Flow Control



Flow control refers to a set of procedures used to restrict the amount of data that the sender can send before waiting for acknowledgment.

# Taxonomy of protocols



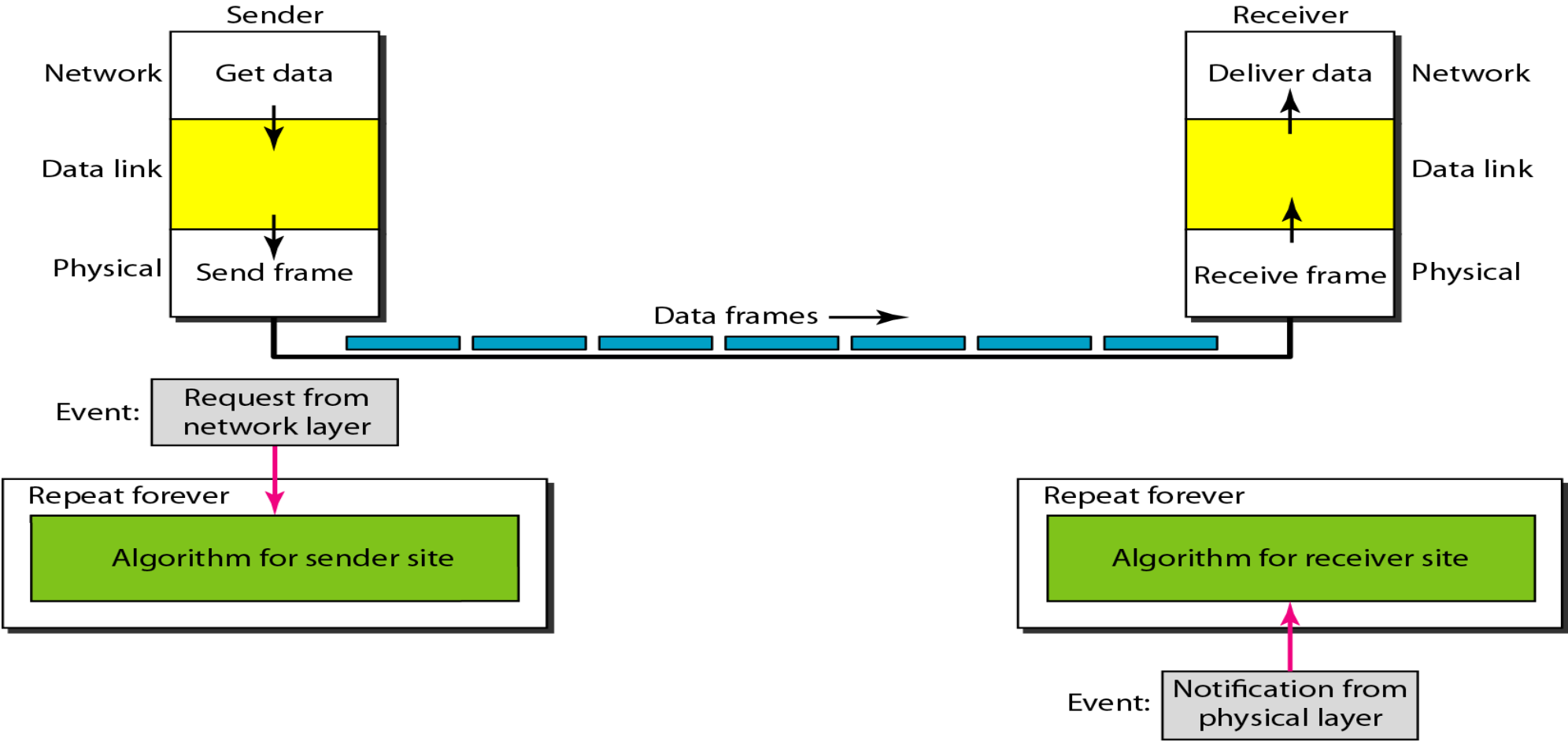
# Flow Control Protocols

- The protocols in the first category(**noiseless**) cannot be used in real life, but they serve as a basis for understanding the protocols of noisy channels.
- All the protocols are unidirectional in the sense that the data frames travel from one node, called the sender, to another node, called the receiver.
- Although special frames, called acknowledgment (**ACK**) and negative acknowledgment (**NAK**) can flow in the opposite direction for flow and error control purposes, data flow in only one direction.
- In a real-life network, the data link protocols are implemented as bidirectional; data flow in both directions.
- In these protocols the flow and error control information such as ACKs and NAKs is included in the data frames in a technique called piggybacking.

# Simplest Protocol

- We divide the discussion of protocols into those that can be used for noiseless (error-free) channels and those that can be used for noisy (error-creating) channels.
- The protocols in the first category cannot be used in real life, but they serve as a basis for understanding the protocols of noisy channels

# Simplest Protocol with no Flow or Error control



# Simplest Protocol with no Flow or Error control

Algorithm *Sender-site algorithm for the simplest protocol*

```
1 while(true)                                // Repeat forever
2 {
3     WaitForEvent();                          // Sleep until an event occurs
4     if(Event(RequestToSend))                //There is a packet to send
5     {
6         GetData();
7         MakeFrame();
8         SendFrame();                          //Send the frame
9     }
10 }
```

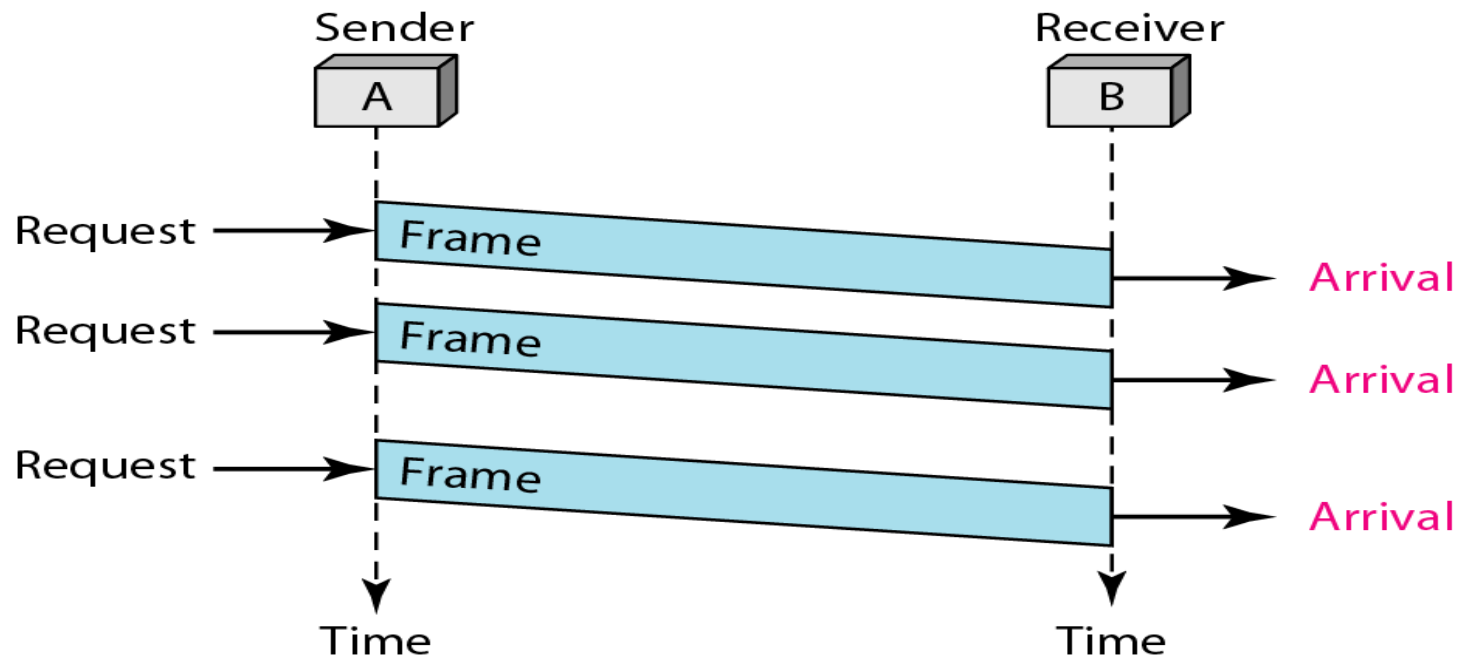
Algorithm *Receiver-site algorithm for the simplest protocol*

```
1 while(true)                                // Repeat forever
2 {
3     WaitForEvent();                          // Sleep until an event occurs
4     if(Event(ArrivalNotification))          //Data frame arrived
5     {
6         ReceiveFrame();
7         ExtractData();
8         DeliverData();                       //Deliver data to network layer
9     }
10 }
```



# Simplest Protocol

To send three frames, three events occur at the sender site and three events at the receiver site.



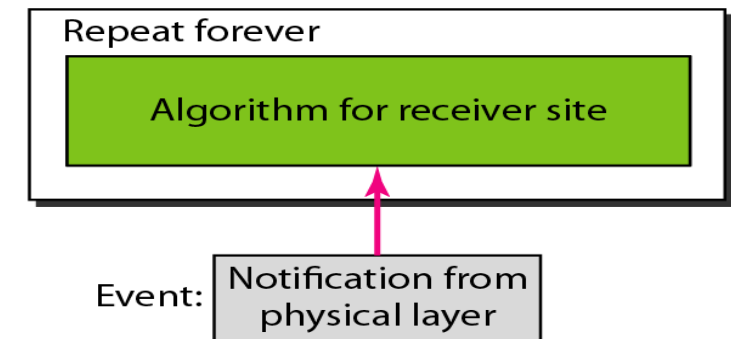
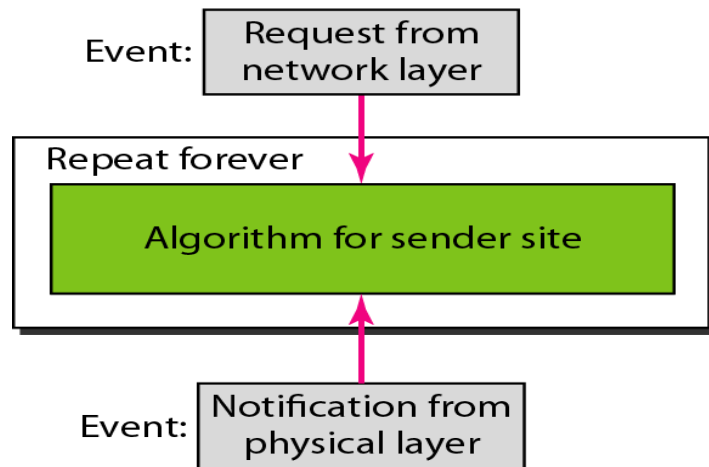
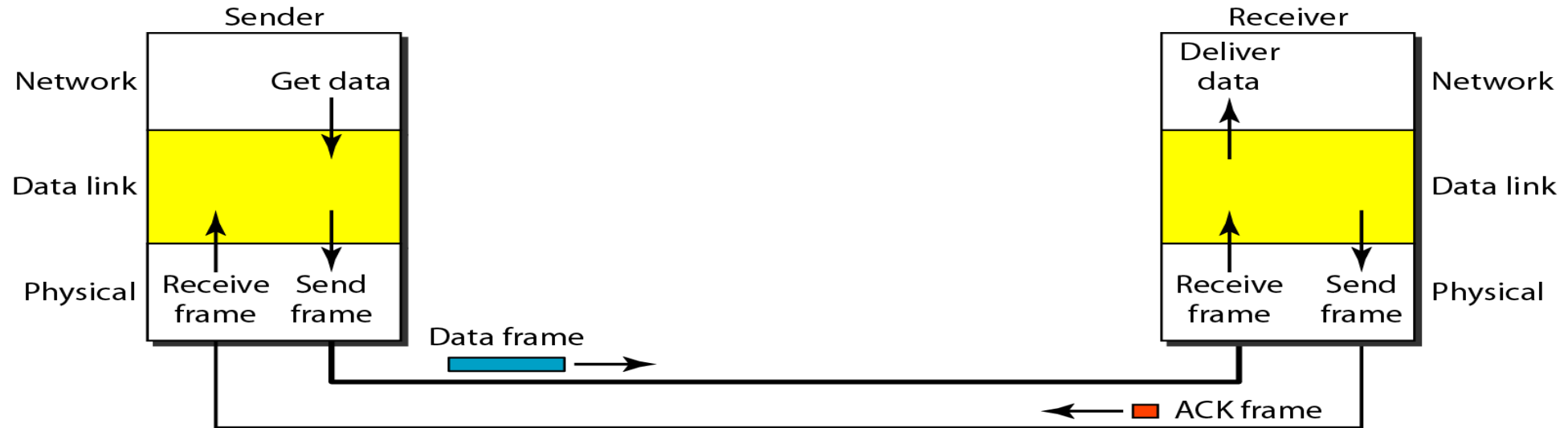
# Problem with Simplest Protocol

- If data frames arrive at the receiver site faster than they can be processed, the frames must be stored until their use.
- Normally, the receiver does not have enough storage space, especially if it is receiving data from many sources.
- This may result in either the discarding of frames or denial of service.
- To prevent the receiver from becoming overwhelmed with frames, we somehow need to tell the sender to slow down.
- There must be feedback from the receiver to the sender

# Stop and Wait Protocol

- Sender sends one frame, stops until it receives confirmation from the receiver (okay to go ahead), and then sends the next frame.
- We still have unidirectional communication for data frames, but auxiliary ACK frames (simple tokens of acknowledgment) travel from the other direction.
- We add flow control to simplest protocol.

# Stop and Wait Protocol



# Stop and Wait Protocol

Algorithm *Sender-site algorithm for Stop-and-Wait Protocol*

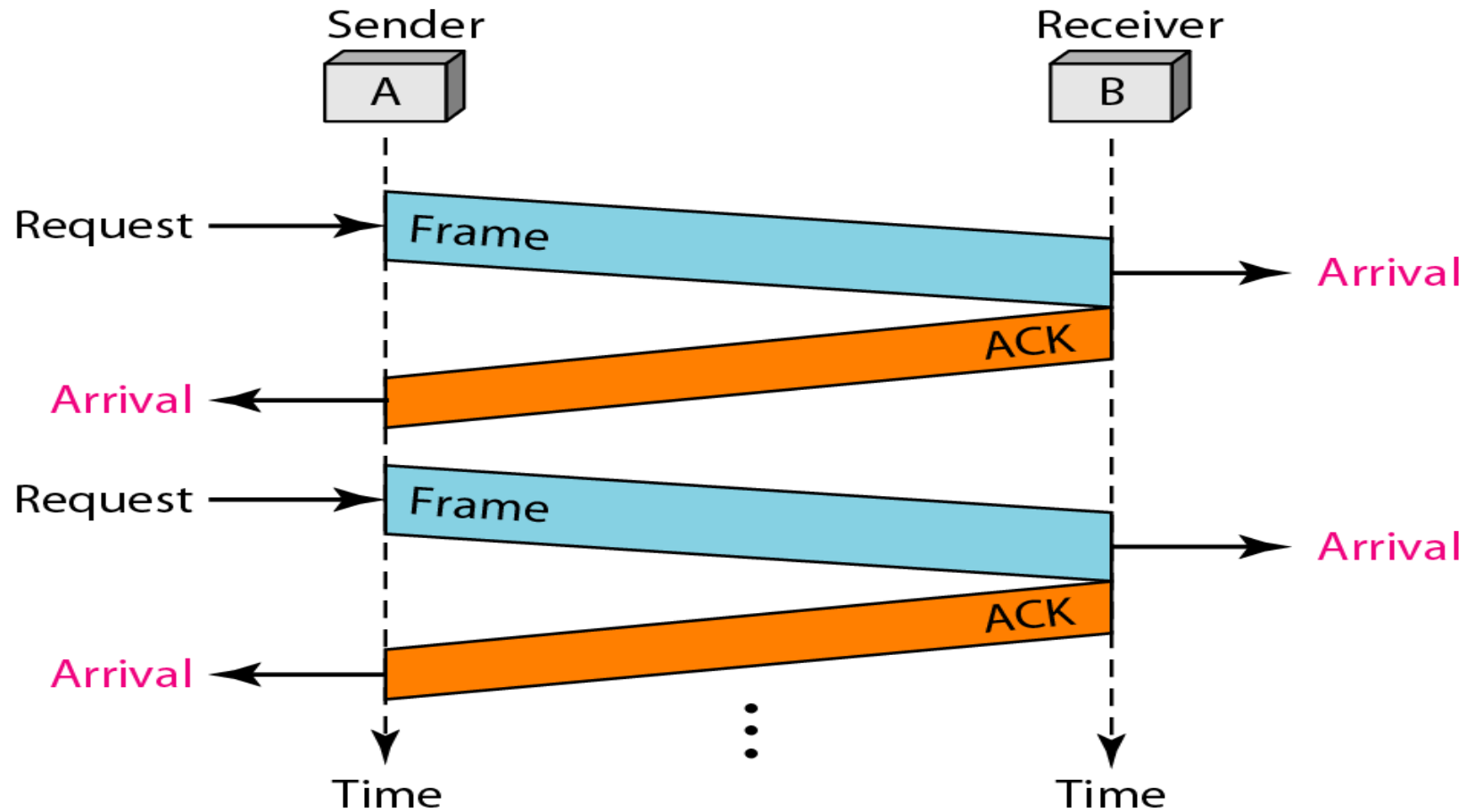
```
1 while(true)                                //Repeat forever
2   canSend = true                            //Allow the first frame to go
3   {
4     WaitForEvent();                        // Sleep until an event occurs
5     if(Event(RequestToSend) AND canSend)
6     {
7       GetData();
8       MakeFrame();
9       SendFrame();                        //Send the data frame
10      canSend = false;                    //Cannot send until ACK arrives
11    }
12    WaitForEvent();                        // Sleep until an event occurs
13    if(Event(ArrivalNotification) // An ACK has arrived
14    {
15      ReceiveFrame();                      //Receive the ACK frame
16      canSend = true;
17    }
18  }
```

# Stop and Wait Protocol

**Algorithm** *Receiver-site algorithm for Stop-and-Wait Protocol*

```
1 while (true)                                //Repeat forever
2 {
3     WaitForEvent();                          // Sleep until an event occurs
4     if (Event (ArrivalNotification))         //Data frame arrives
5     {
6         ReceiveFrame();
7         ExtractData();
8         Deliver(data);                       //Deliver data to network layer
9         SendFrame();                         //Send an ACK frame
10    }
11 }
```

# Stop and Wait Protocol



# Noisy Channels

- Although the Stop-and-Wait Protocol gives us an idea of how to add flow control to its predecessor, noiseless channels are nonexistent. We discuss three protocols in this section that use error control.

## Noisy Channels Flow control Protocols

- Stop-and-Wait Automatic Repeat Request
- Go-Back-N Automatic Repeat Request
- Selective Repeat Automatic Repeat Request



# Stop and Wait Automatic Repeat Request (1 bit protocol)

- The Error control mechanism added in Stop and Wait Protocol.
- To detect and correct corrupted frames, we need to add **redundancy bits** to our data frame.
- Lost frames are more difficult to handle than corrupted ones. In our previous protocols, there was no way to identify a frame.
- The received frame could be the correct one, or a duplicate, or a frame out of order. The solution is to number the frames.
- When the receiver receives a data frame that is out of order, this means that frames were either **lost** or **duplicated**

# Stop and Wait Automatic Repeat Request (1 bit protocol)

Error correction in Stop-and-Wait ARQ is done by keeping a copy of the sent frame and retransmitting of the frame when the timer expires.

# Stop and Wait Automatic Repeat Request

- If the timer expires and there is no ACK for the sent frame, the frame is resent, the copy is held, and the timer is restarted.
- Since an ACK frame can also be corrupted and lost, it too needs redundancy bits and a sequence number.
- The ACK frame for this protocol has a sequence number field.
- In this protocol, the sender simply discards a corrupted ACK frame or ignores an out-of-order one.

# Stop and Wait Automatic Repeat Request

## Sequence Number :

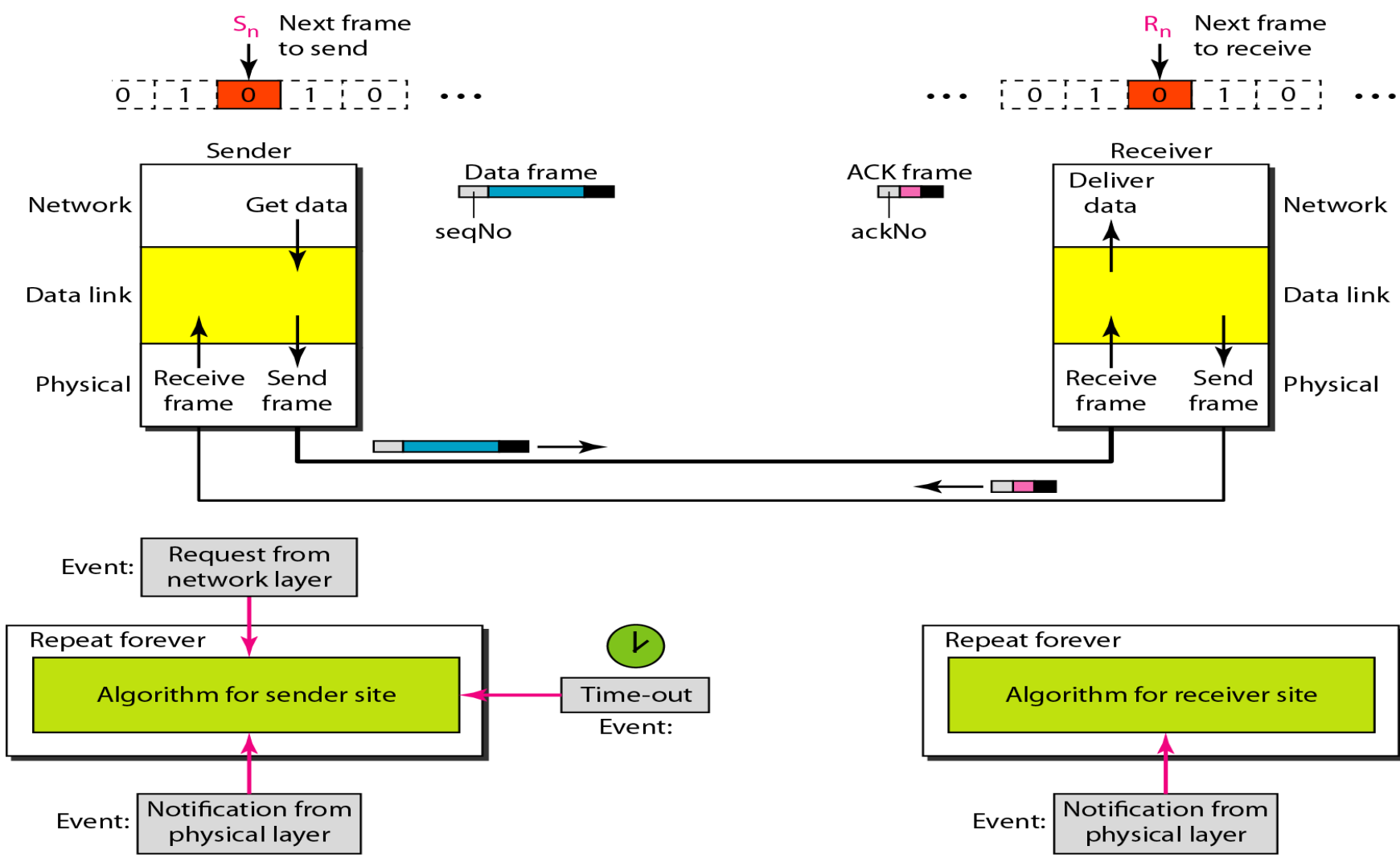
- A field is added to the data frame to hold the sequence number of that frame.
- We want to minimize the frame size, we look smallest range that provides unambiguous communication.

In Stop-and-Wait ARQ, we use sequence numbers to number the frames.  
The sequence numbers are based on modulo-2 arithmetic.

In Stop-and-Wait ARQ, the acknowledgment number always announces in modulo-2 arithmetic the sequence number of the next frame expected.

# Stop and Wait Automatic Repeat Request

Design of  
Stop-and-wait ARQ



# Stop and Wait Automatic Repeat Request

Algorithm *Sender-site algorithm for Stop-and-Wait ARQ*

```
1  Sn = 0;                                // Frame 0 should be sent first
2  canSend = true;                          // Allow the first request to go
3  while(true)                              // Repeat forever
4  {
5      WaitForEvent();                      // Sleep until an event occurs
6      if(Event(RequestToSend) AND canSend)
7      {
8          GetData();
9          MakeFrame(Sn);                  //The seqNo is Sn
10         StoreFrame(Sn);                //Keep copy
11         SendFrame(Sn);
12         StartTimer();
13         Sn = Sn + 1;
14         canSend = false;
15     }
16     WaitForEvent();                      // Sleep
```

(continued)

# Stop and Wait Automatic Repeat Request

```
17     if (Event (ArrivalNotification)           // An ACK has arrived
18     {
19         ReceiveFrame (ackNo) ;                 //Receive the ACK frame
20         if(not corrupted AND ackNo == Sn) //Valid ACK
21         {
22             Stoptimer() ;
23             PurgeFrame (Sn-1) ;                //Copy is not needed
24             canSend = true;
25         }
26     }
27
28     if (Event (TimeOut)                       // The timer expired
29     {
30         StartTimer() ;
31         ResendFrame (Sn-1) ;                 //Resend a copy check
32     }
33 }
```

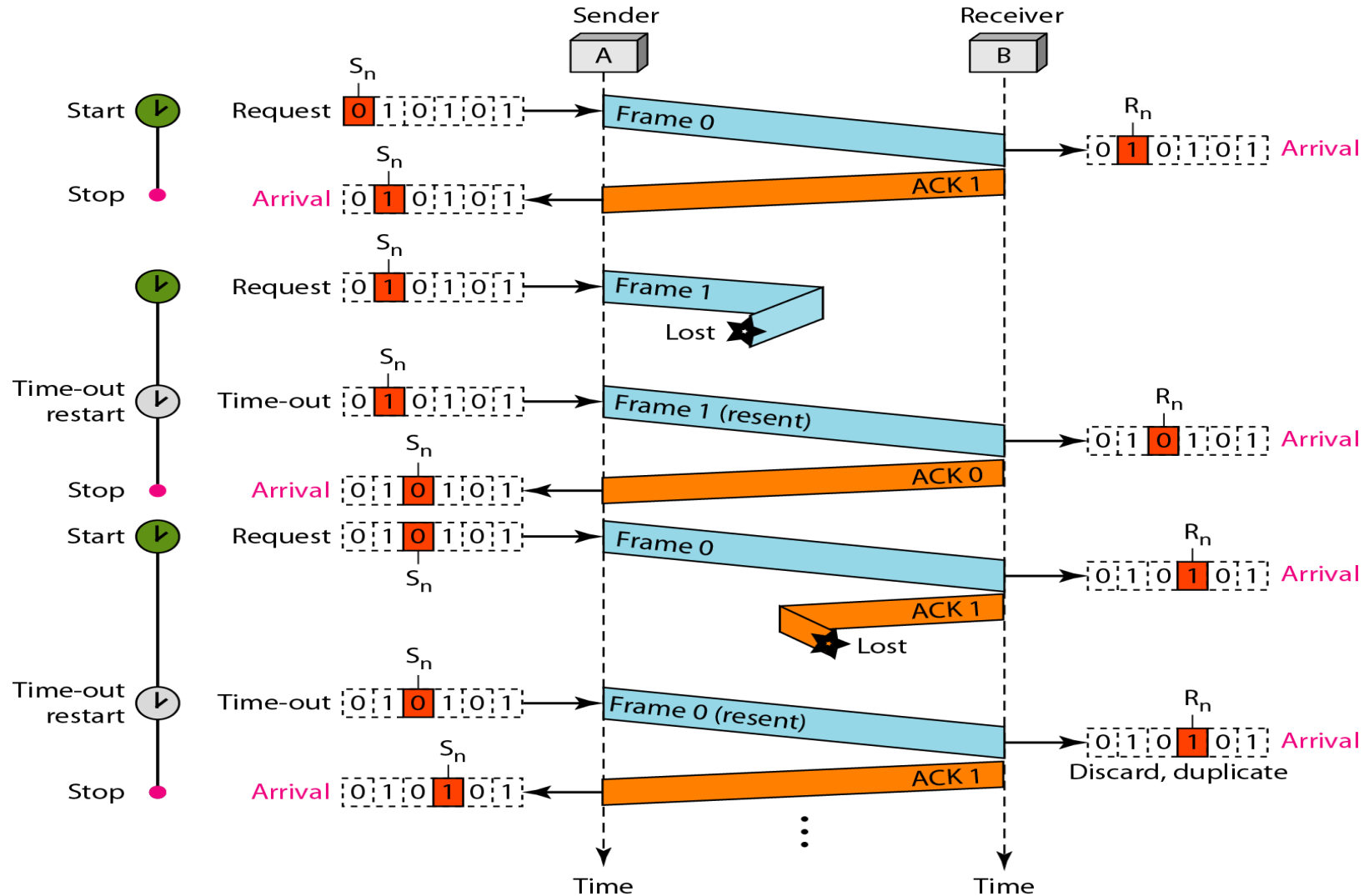
# Stop and Wait Automatic Repeat Request

**Algorithm** *Receiver-site algorithm for Stop-and-Wait ARQ Protocol*

```
1  Rn = 0;                                // Frame 0 expected to arrive first
2  while(true)
3  {
4      WaitForEvent();                      // Sleep until an event occurs
5      if(Event(ArrivalNotification))      //Data frame arrives
6      {
7          ReceiveFrame();
8          if(corrupted(frame));
9              sleep();
10         if(seqNo == Rn)                  //Valid data frame
11         {
12             ExtractData();
13             DeliverData();                //Deliver data
14             Rn = Rn + 1;
15         }
16         SendFrame(Rn);                  //Send an ACK
17     }
18 }
```



# Stop and Wait ARQ - Example



# Stop and Wait ARQ - Efficiency

- The Stop-and-WaitARQ is very inefficient if our channel is thick and long. By thick, we mean that our channel has a large bandwidth; by long,
- we mean the round-trip delay is long. The product of these two is called the bandwidth delay product(BDP).
- We can think of the channel as a pipe. The
- bandwidth-delay product then is the volume of the pipe in bits. The pipe is always there.
- If we do not use it, we are inefficient. The bandwidth-delay product is a measure of the number of bits we can send out of our system while waiting for news from the receive.

### Example 11.4

*Assume that, in a Stop-and-Wait ARQ system, the bandwidth of the line is 1 Mbps, and 1 bit takes 20 ms to make a round trip. What is the bandwidth-delay product? If the system data frames are 1000 bits in length, what is the utilization percentage of the link?*

### Solution

The bandwidth-delay product is

$$(1 \times 10^6) \times (20 \times 10^{-3}) = 20,000 \text{ bits}$$



### Example 11.4 (continued)

*The system can send 20,000 bits during the time it takes for the data to go from the sender to the receiver and then back again.*

*However, the system sends only 1000 bits.*

*We can say that the link utilization is only  $1000/20,000$ , or 5 percent.*

*For this reason, for a link with a high bandwidth or long delay, the use of Stop-and-Wait ARQ wastes the capacity of the link.*



### Example 11.5

*What is the utilization percentage of the link in Example 11.4 if we have a protocol that can send up to 15 frames before stopping and worrying about the acknowledgments?*

### *Solution*

*The bandwidth-delay product is still 20,000 bits. The system can send up to 15 frames or 15,000 bits during a round trip.*

*This means the utilization is  $15,000/20,000$ , or 75 percent. Of course, if there are damaged frames, the utilization percentage is much less because frames have to be resent.*

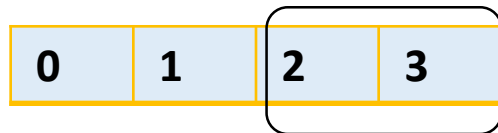
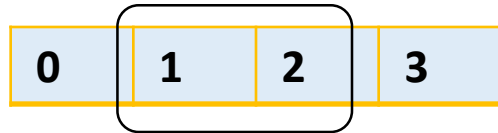
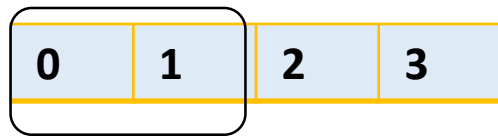
# Sliding window protocol

- Sequence number is given to each out bound frame from 0 to N.
- If the  $m$  bits allow for sequence number , the sequence number from 0 to  $2^m - 1$ .
- Sliding window : these are the imaginary boxes of transmitter and receiver ( buffer).

# Basic Sliding window protocol

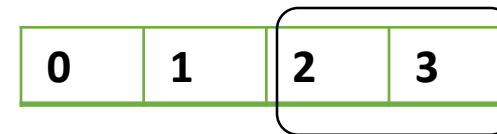
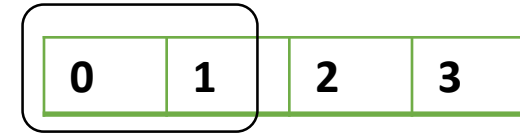
- Sliding window size is 2

Sender

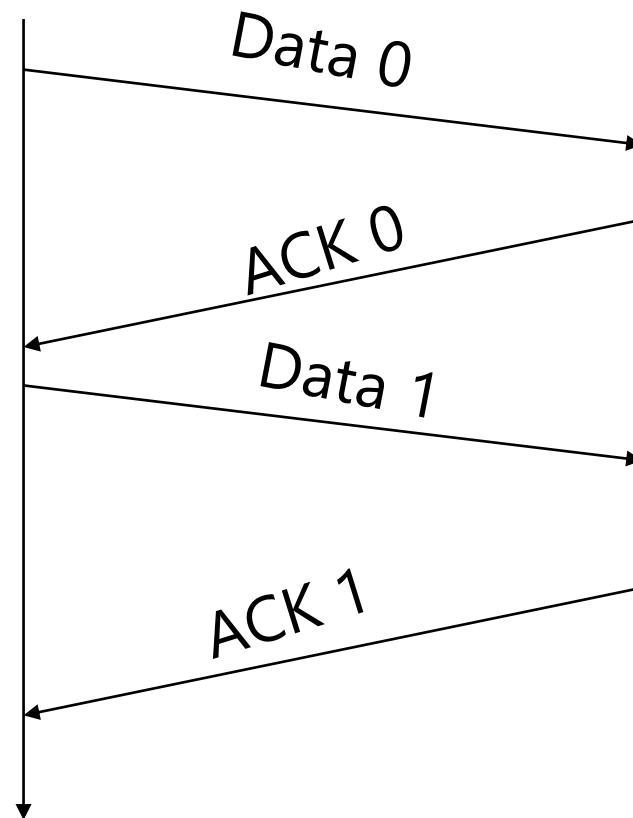


Move right, when ACK is received

Receiver

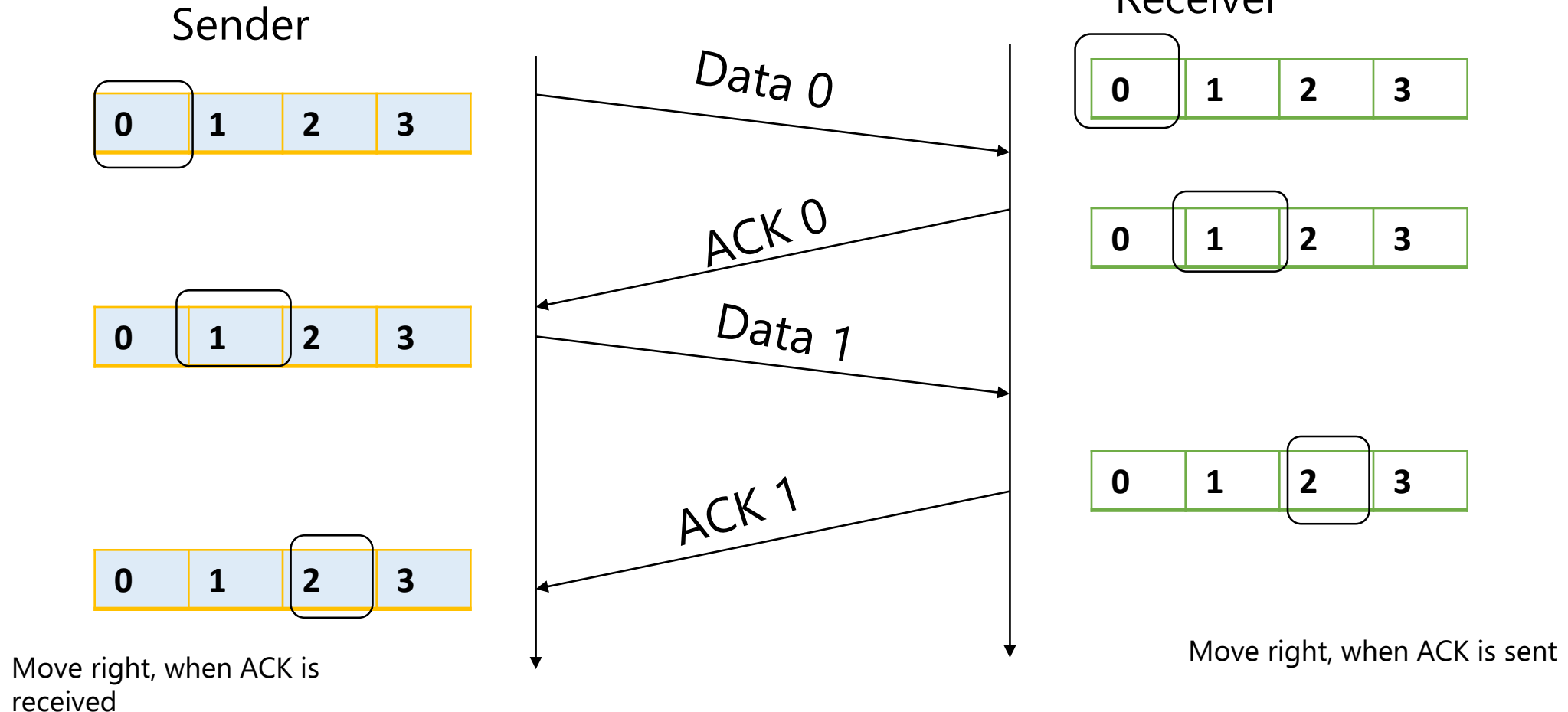


Move right, when ACK is sent



# Stop and Wait ARQ ( with sliding window)

- Maximum window size is 1 ( one bit protocol)





# Go-Back-N ARQ ( Automatic Repeat Request) Protocol

- To improve the efficiency of transmission (filling the pipe), multiple frames must be in transition while waiting for acknowledgment.
- Keep copy of frames until the acknowledgment arrive.
- Sender windows size is **N** and receiver window size is **1**
- If the  $m$  bits allow for sequence number , the sequence number from 0 to  $2^m - 1$ .

if  $m=2$  then frame number from 0 to 3

*0,1,2,3, 0,1,2,3, 0,1,2,3, 0,1,2,3, 0,1,2,3, 0,1,2,3,*

# Go-Back-N ARQ ( Automatic Repeat Request) Protocol

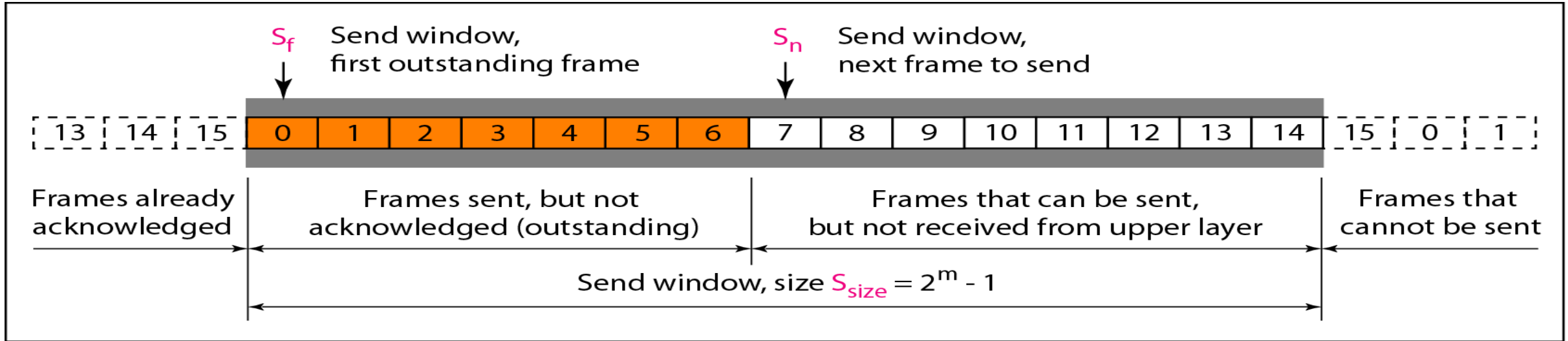
In the Go-Back-N Protocol, the sequence numbers are modulo  $2^m$ , where  $m$  is the size of the sequence number field in bits.

# Go-Back-N ARQ ( Automatic Repeat Request) Protocol

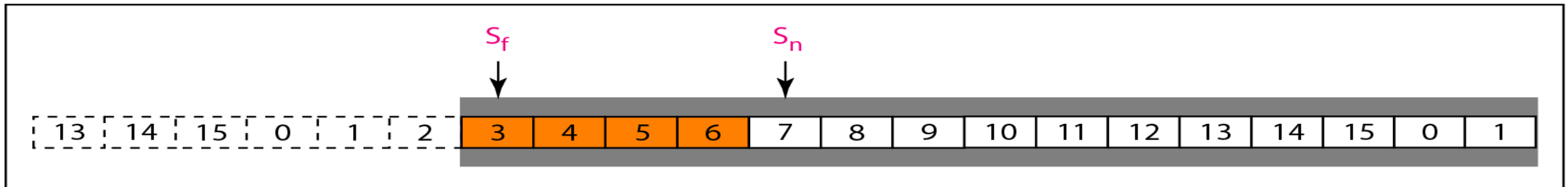
## Sliding Window

- Sliding window define the range of sequence numbers that is concern of the sender and receiver.
- The range which is the concern of the sender is called the ***send sliding window***
- The range that is the concern of the receiver is called the ***receive sliding window.***
- consider window size is  $m = 4$  , so frame sequence number is from 0 to 15.
- The window at any time divides the possible sequence numbers into four regions.

# Send window for Go-Back-N ARQ



a. Send window before sliding

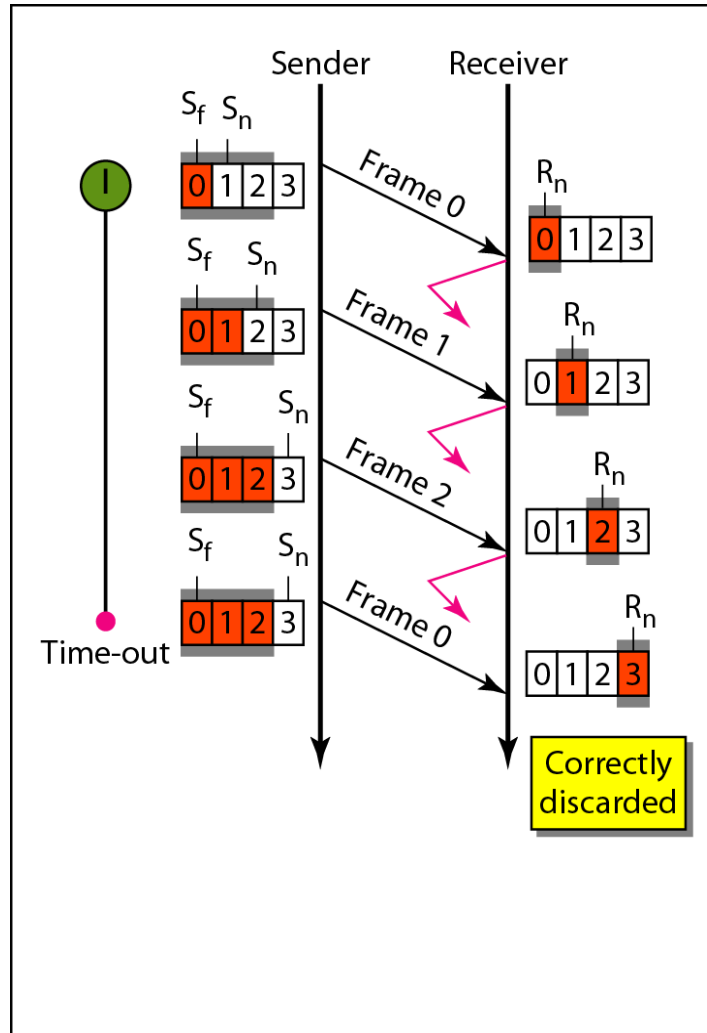


b. Send window after sliding

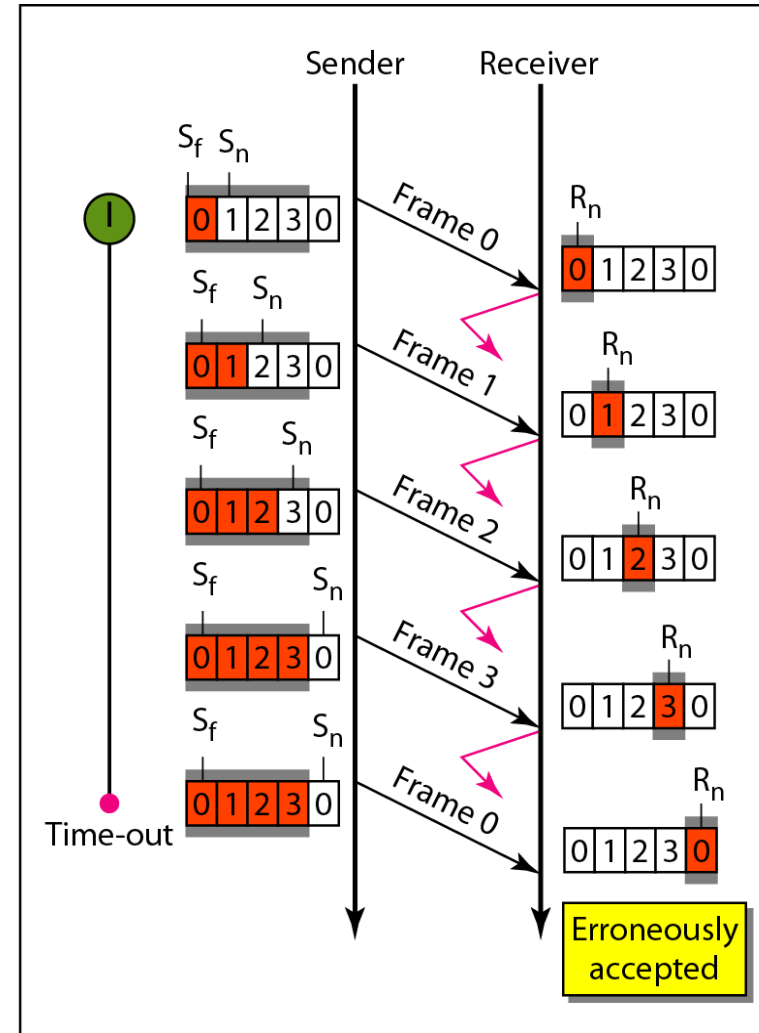
# Send window for Go-Back-N ARQ

The send window is an abstract concept defining an imaginary box of size  $2^m - 1$  with three variables: Sf, Sn, and S<sub>size</sub>.

# Window size for Go-Back-N ARQ



a. Window size  $< 2^m$

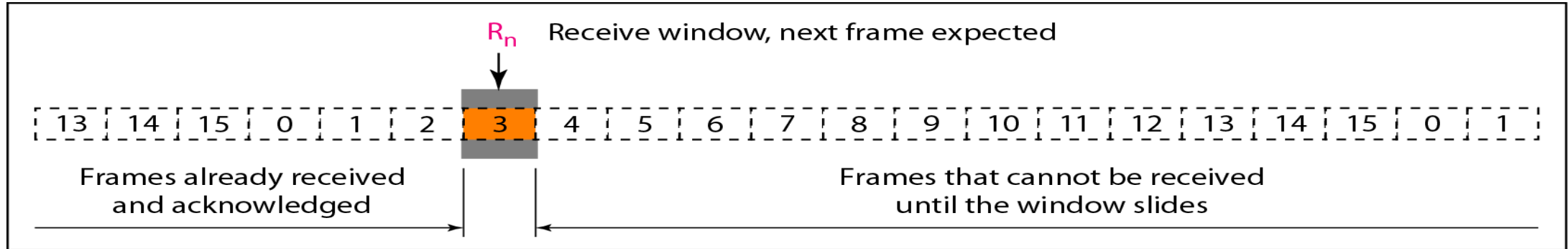


b. Window size  $= 2^m$

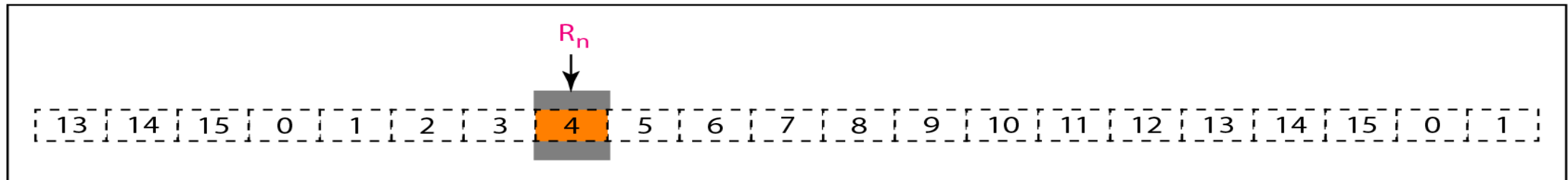
# Send window for Go-Back-N ARQ

The send window can slide one or more slots when a valid acknowledgment arrives.

# Receiver window for Go-Back-N ARQ



a. Receive window



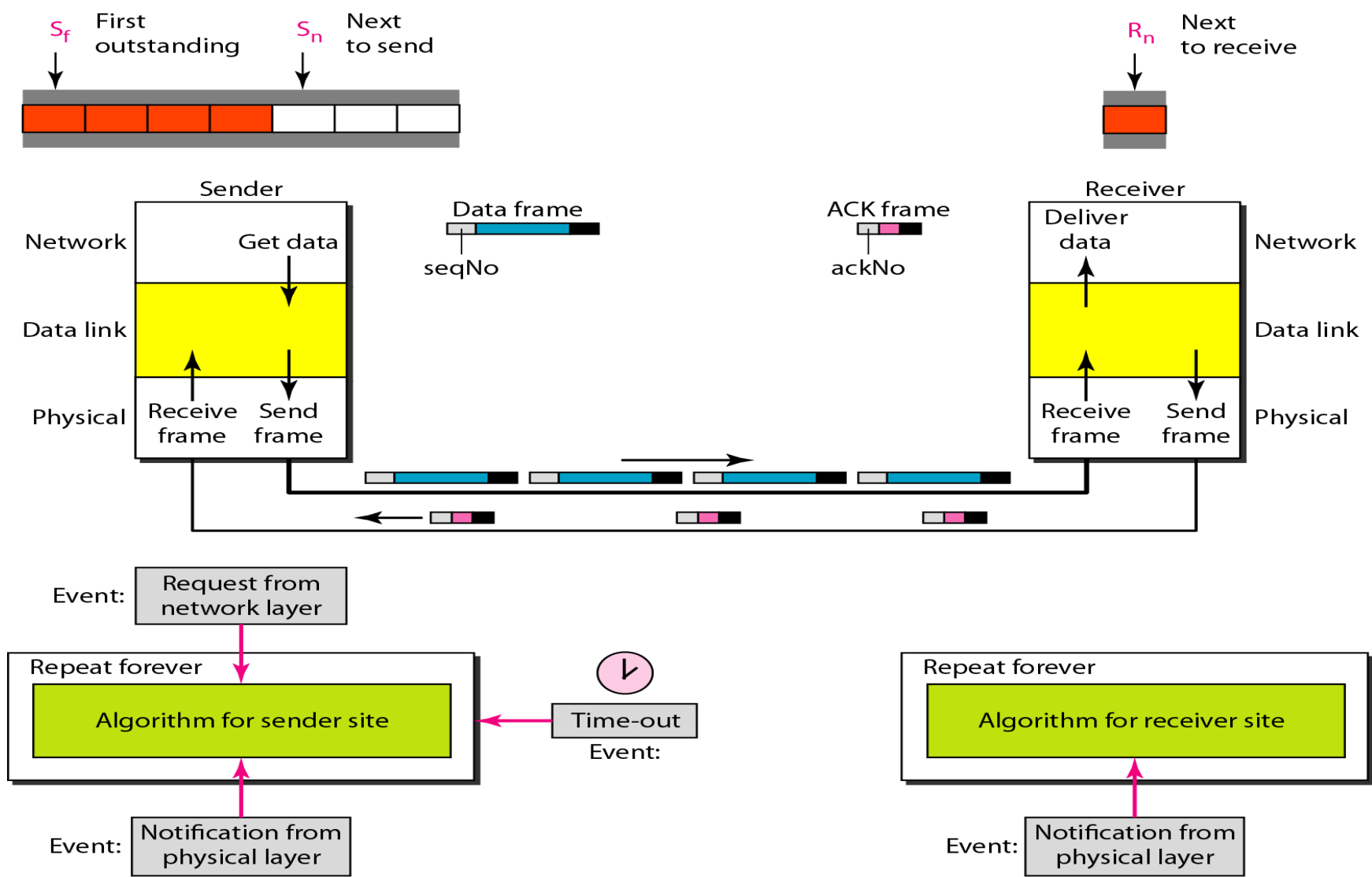
b. Window after sliding



# Receiver window for Go-Back-N ARQ

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

# Design of Go-Back-N ARQ





## *Note*

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

## Go-Back-N sender algorithm

```
1  Sw = 2m - 1;
2  Sf = 0;
3  Sn = 0;
4
5  while (true)                                //Repeat forever
6  {
7      WaitForEvent();
8      if (Event (RequestToSend))              //A packet to send
9      {
10         if (Sn - Sf >= Sw)                 //If window is full
11             Sleep();
12         GetData();
13         MakeFrame (Sn);
14         StoreFrame (Sn);
15         SendFrame (Sn);
16         Sn = Sn + 1;
17         if (timer not running)
18             StartTimer();
19     }
20
```

(continued)

```
21  if (Event (ArrivalNotification))  //ACK arrives
22  {
23      Receive (ACK);
24      if (corrupted (ACK))
25          Sleep();
26      if ((ackNo > Sf) && (ackNo ≤ Sn))  //If a valid ACK
27      While (Sf ≤ ackNo)
28      {
29          PurgeFrame (Sf);
30          Sf = Sf + 1;
31      }
32      StopTimer();
33  }
34
35  if (Event (TimeOut))  //The timer expires
36  {
37      StartTimer();
38      Temp = Sf;
39      while (Temp < Sn);
40      {
41          SendFrame (Sf);
42          Sf = Sf + 1;
43      }
44  }
45 }
```

## Go-Back-N receiver algorithm

```
1  Rn = 0;
2
3  while (true)                                //Repeat forever
4  {
5      WaitForEvent();
6
7      if(Event(ArrivalNotification))          //Data frame arrives
8      {
9          Receive(Frame);
10         if(corrupted(Frame))
11             Sleep();
12         if(seqNo == Rn)                      //If expected frame
13         {
14             DeliverData();                  //Deliver data
15             Rn = Rn + 1;                    //Slide window
16             SendACK(Rn);
17         }
18     }
19 }
```



### *Note*

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

# Go-Back-N

- Sliding window size is 4, ACK is lost

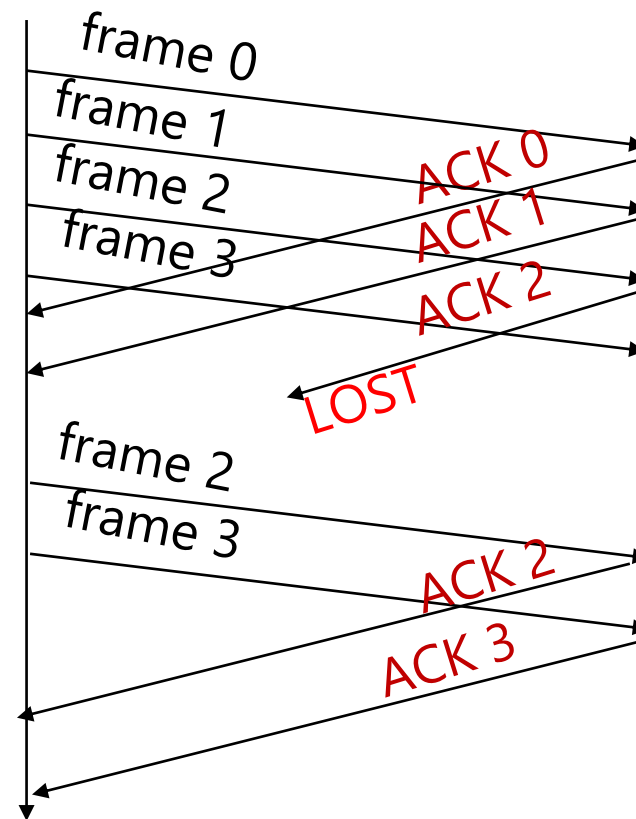


Sender

Receiver



Timer



Frame Discarded





# Go-Back-N

- Sliding window size is 4, Frame lost/damaged

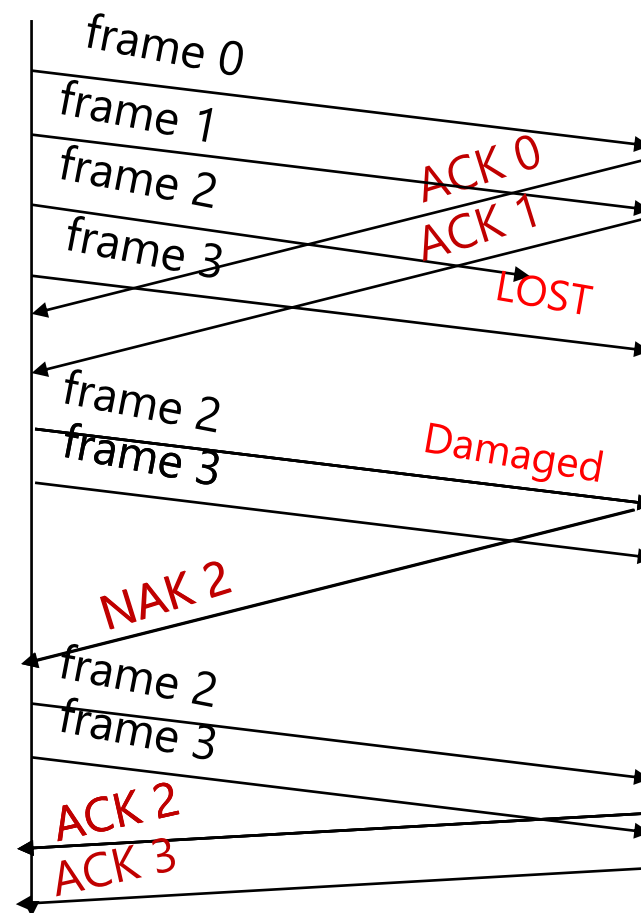


Sender

Receiver



Timer

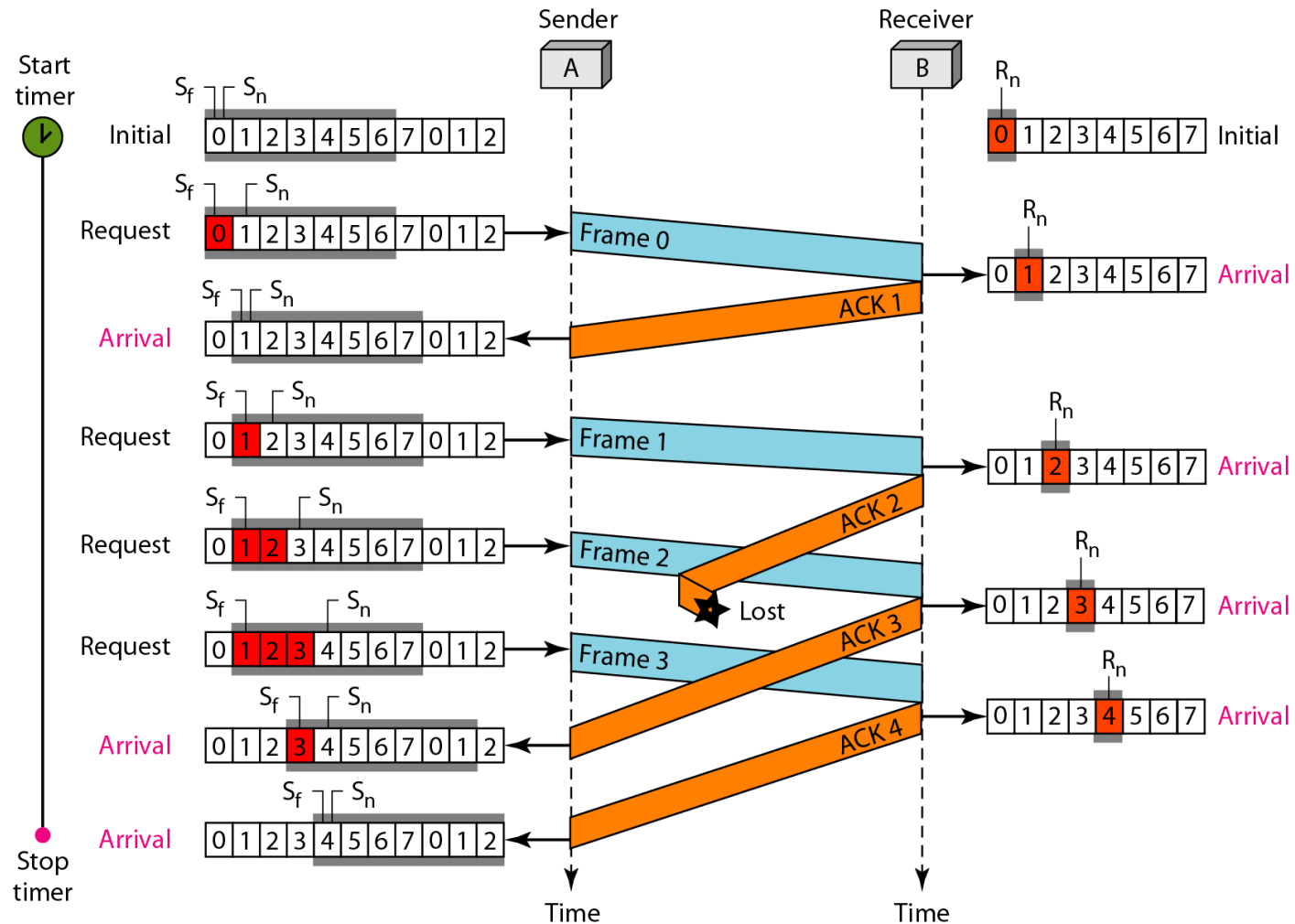


Frame Discarded

Frame Discarded



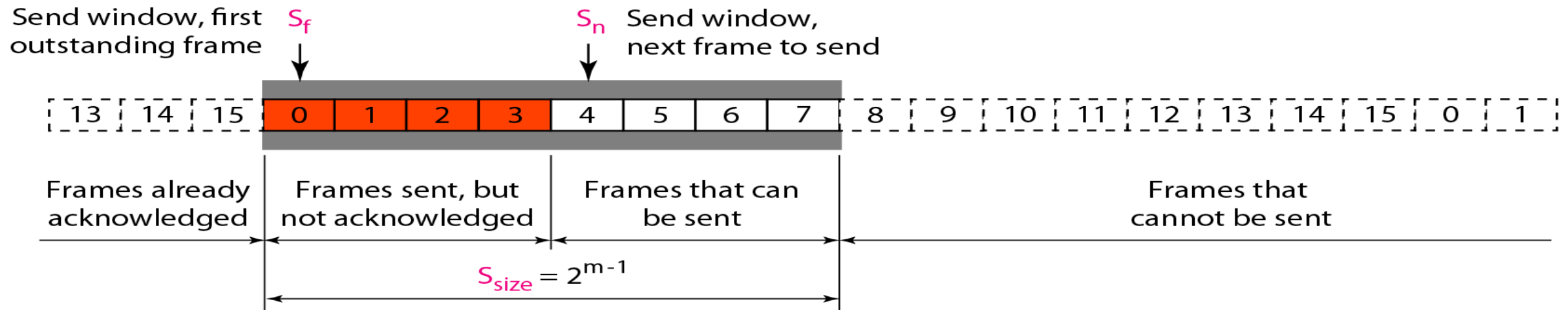
# Go-Back-N ARQ



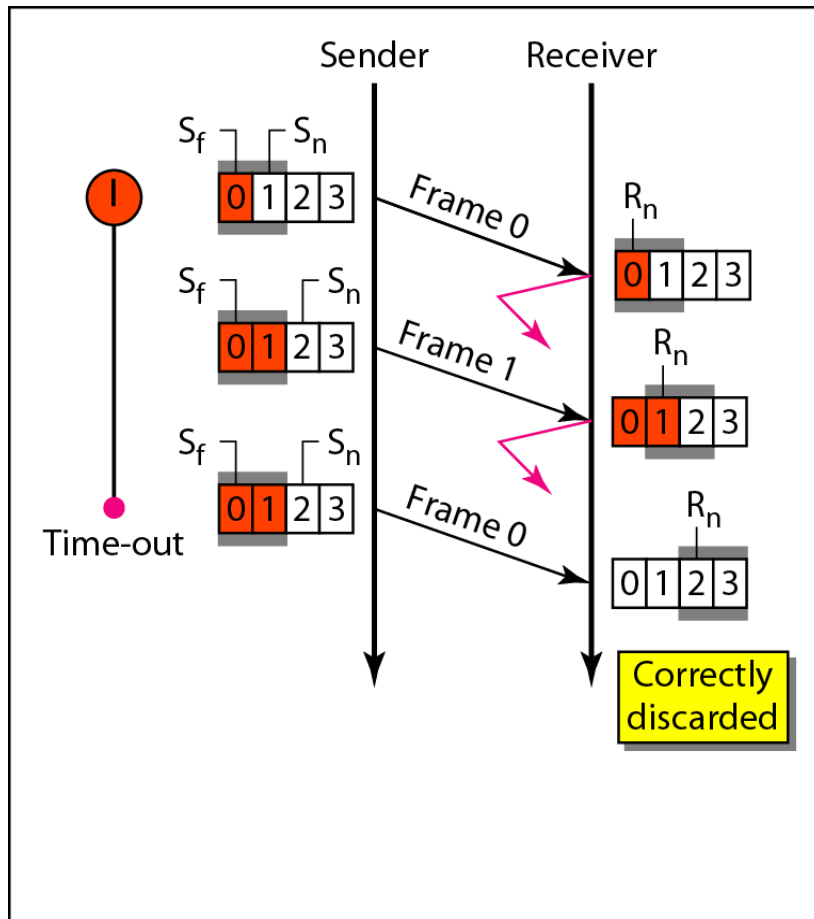
# Selective Repeat – ARQ ( Automatic Repeat Request) Protocol

- In Go-Back-N , the Sender windows size is  $N$  and receiver window size is  $1$ . so at receiver there is no need to buffer out-of-order frames.
- However, Go-Back-N is very inefficient for a noisy link. In a noisy link a frame has a higher probability of damage, which means the resending of multiple frames.
- This resending uses up the bandwidth and slows down the transmission. For noisy links, there is another mechanism that does not resend  $N$  frames when just one frame is damaged; only the damaged frame is resent. This mechanism is called Selective Repeat ARQ.
- It is more efficient for noisy links, but the processing at the receiver is more complex.
- In Selective Repeat ARQ , the Sender windows size is  $N$  and receiver window size is  $N$ .

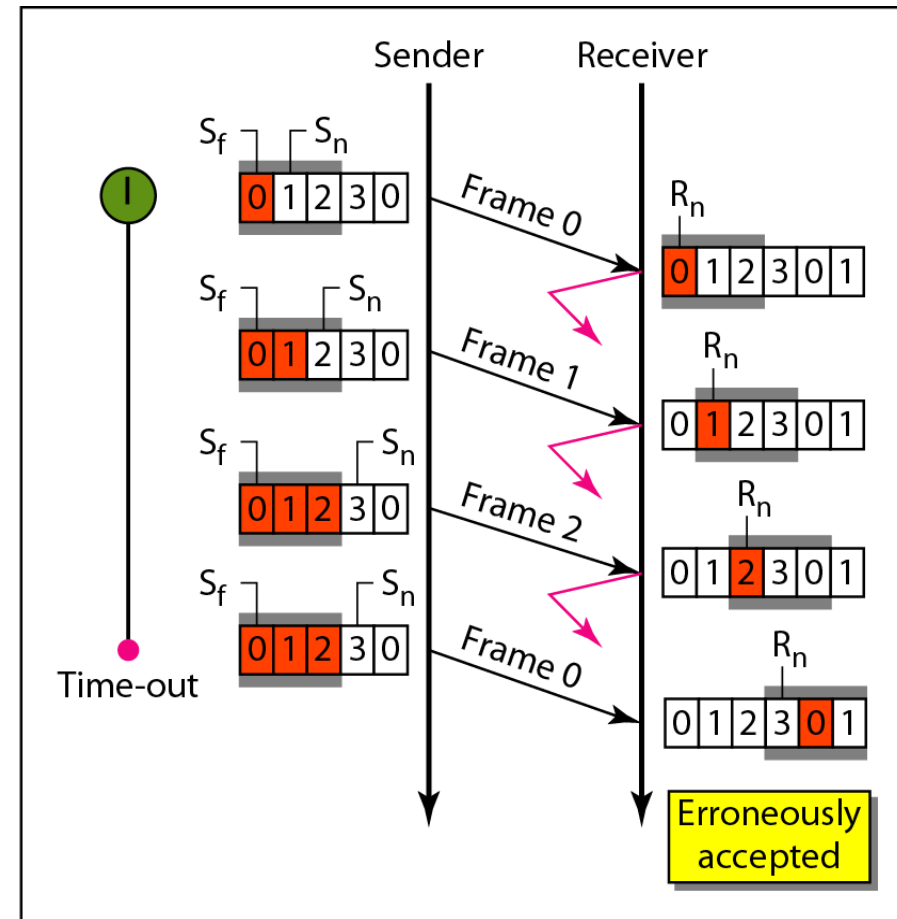
# Send window for Selective Repeat ARQ



# Window Size for Selective Repeat ARQ

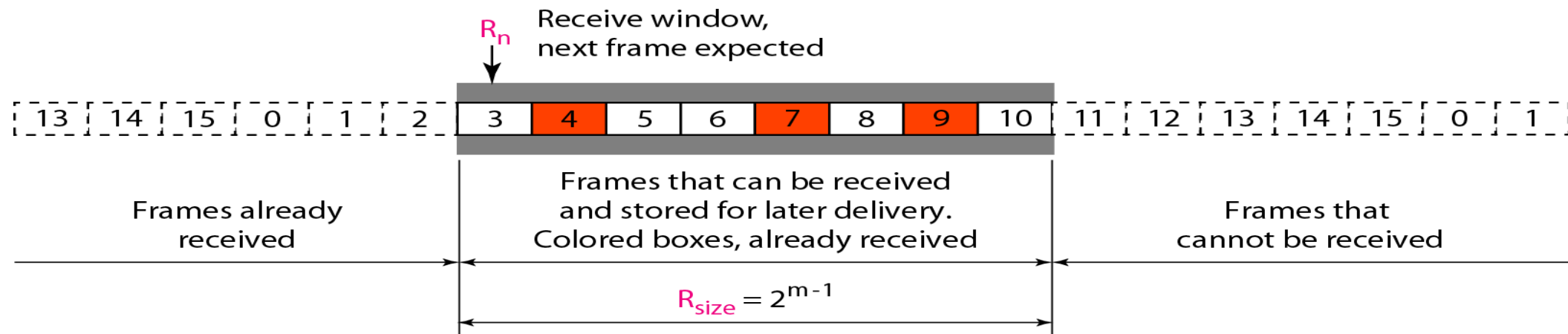


a. Window size =  $2^{m-1}$

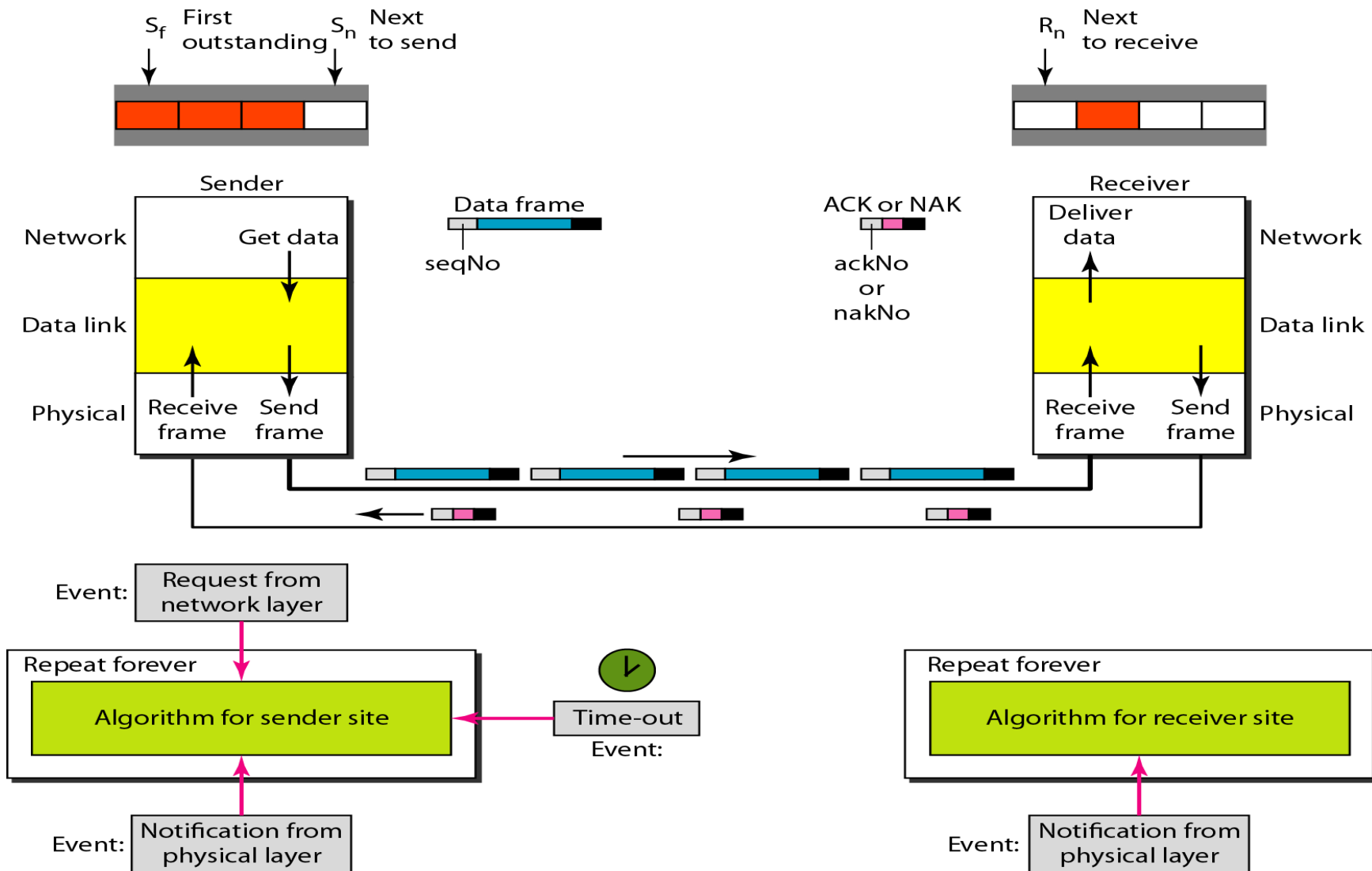


b. Window size >  $2^{m-1}$

# Receive window for Selective Repeat ARQ



# Design of Selective Repeat ARQ



## Sender-site Selective Repeat algorithm

```
1   $S_w = 2^{m-1}$  ;
2   $S_f = 0$  ;
3   $S_n = 0$  ;
4
5  while (true)                                //Repeat forever
6  {
7      WaitForEvent() ;
8      if (Event (RequestToSend) )             //There is a packet to send
9      {
10         if ( $S_n - S_f \geq S_w$ )                //If window is full
11             Sleep() ;
12         GetData() ;
13         MakeFrame ( $S_n$ ) ;
14         StoreFrame ( $S_n$ ) ;
15         SendFrame ( $S_n$ ) ;
16          $S_n = S_n + 1$  ;
17         StartTimer ( $S_n$ ) ;
18     }
19
```

(continued)



## Sender-site Selective Repeat algorithm

(continued)

```
20  if (Event (ArrivalNotification)) //ACK arrives
21  {
22      Receive (frame); //Receive ACK or NAK
23      if (corrupted (frame))
24          Sleep();
25      if (FrameType == NAK)
26          if (nakNo between  $S_f$  and  $S_n$ )
27          {
28              resend (nakNo);
29              StartTimer (nakNo);
30          }
31      if (FrameType == ACK)
32          if (ackNo between  $S_f$  and  $S_n$ )
33          {
34              while ( $s_f < \text{ackNo}$ )
35              {
36                  Purge ( $s_f$ );
37                  StopTimer ( $s_f$ );
38                   $S_f = S_f + 1$ ;
39              }
40          }
41  }
```

(continued)

```
42
43  if (Event (TimeOut (t) ) )           //The timer expires
44  {
45      StartTimer (t) ;
46      SendFrame (t) ;
47  }
48 }
```

## Receiver-site Selective Repeat algorithm

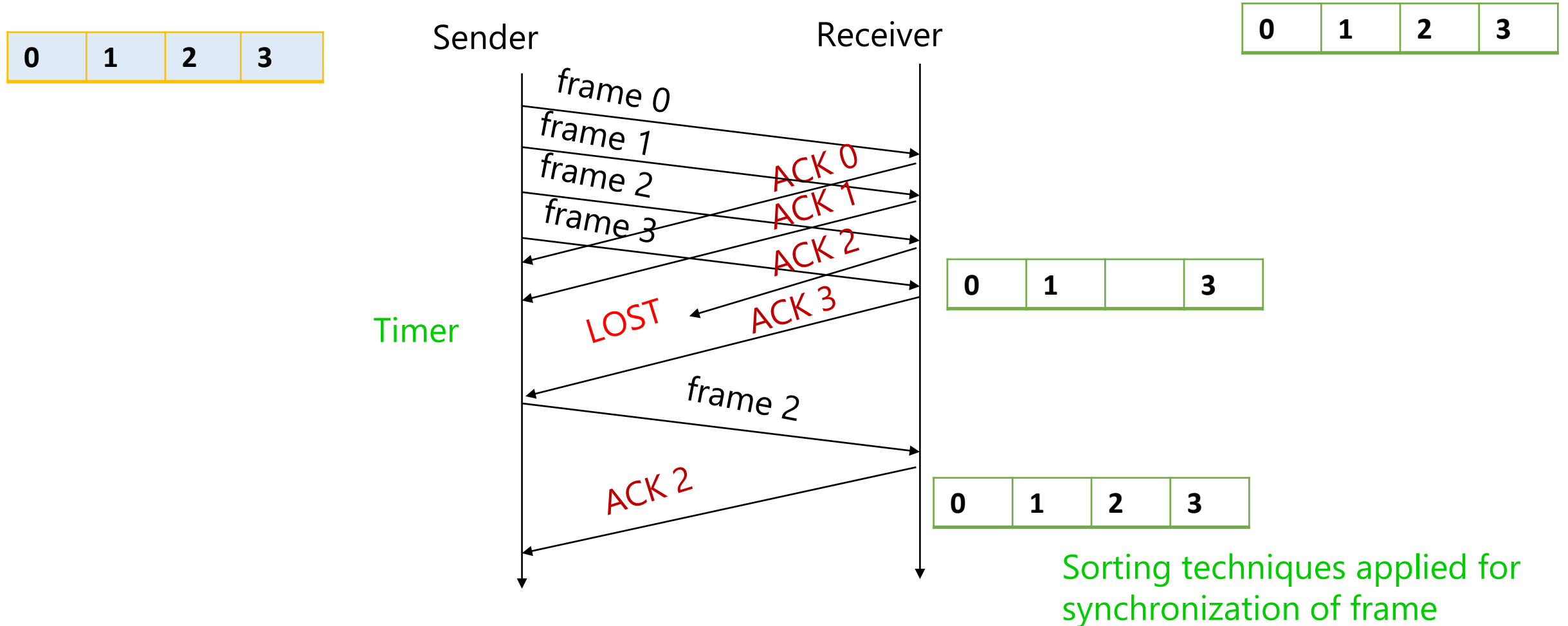
```
1  Rn = 0;
2  NakSent = false;
3  AckNeeded = false;
4  Repeat(for all slots)
5      Marked(slot) = false;
6
7  while (true)                                //Repeat forever
8  {
9      WaitForEvent();
10
11     if(Event(ArrivalNotification))           //Data frame arrives
12     {
13         Receive(Frame);
14         if(corrupted(Frame)) && (NOT NakSent)
15         {
16             SendNAK(Rn);
17             NakSent = true;
18             Sleep();
19         }
20         if(seqNo <> Rn) && (NOT NakSent)
21         {
22             SendNAK(Rn);
```

## Receiver-site Selective Repeat algorithm

```
23     NakSent = true;
24     if ((seqNo in window)&&(!Marked(seqNo)))
25     {
26         StoreFrame(seqNo)
27         Marked(seqNo)= true;
28         while(Marked(Rn))
29         {
30             DeliverData(Rn);
31             Purge(Rn);
32             Rn = Rn + 1;
33             AckNeeded = true;
34         }
35         if(AckNeeded);
36         {
37             SendAck(Rn);
38             AckNeeded = false;
39             NakSent = false;
40         }
41     }
42 }
43 }
44 }
```

# Selective Repeat – ARQ Example

- Sliding window size is 4, ACK is lost



# Selective Repeat – ARQ Example

- Sliding window size is 4, Frame lost/damaged

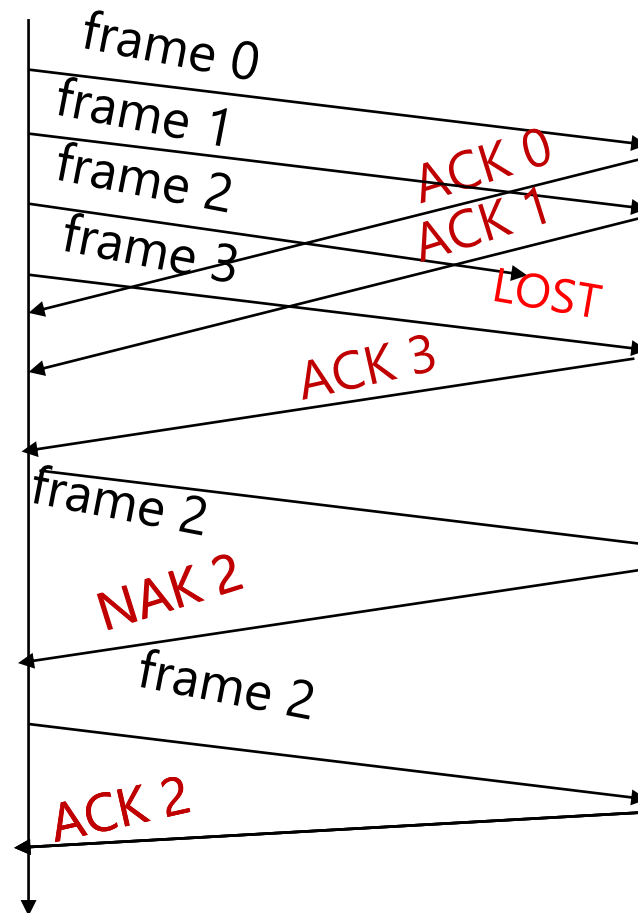
0	1	2	3
---	---	---	---

Sender

Receiver

0	1	2	3
---	---	---	---

Timer



0	1		3
---	---	--	---

Frame Discarded

0	1		3
---	---	--	---

0	1	2	3
---	---	---	---

Sorting techniques  
applied for  
synchronization of frame

# Go- Back –N ARQ and Selective Repeat ARQ

## Go- Back –N ARQ

- Retransmitted rest of 'N' no of frames in case of any problem.
- If error – rate is high, it wastes lot of bandwidth.
- Less complicated.
- Sorting is not require.
- Most often used.

## Selective Repeat ARQ

- Retransmitted only those frames that have problem.
- Less wastage of bandwidth.
- More complicated.
- Sorting is require.
- Rarely used, due to complexity of algorithm.

# piggybacking in Go-Back-N ARQ

- The three protocols we discussed in this section are all unidirectional: data frames flow in only one direction although control information such as ACK and NAK frames can travel in the other direction.
- In real life, data frames are normally flowing in both directions: from node A to node B and from node B to node A.
- This means that the control information also needs to flow in both directions. A technique called piggybacking is used to improve the efficiency of the bidirectional protocols.
- When a frame is carrying data from A to B, it can also carry control information about arrived (or lost) frames from B; when a frame is carrying data from B to A, it can also carry control information about the arrived (or lost) frames from A.



# Design of piggybacking in Go-Back-N ARQ

