Chapter 3- Part4

The Data Link Layer
Sliding Windows Protocols

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

Piggybacking

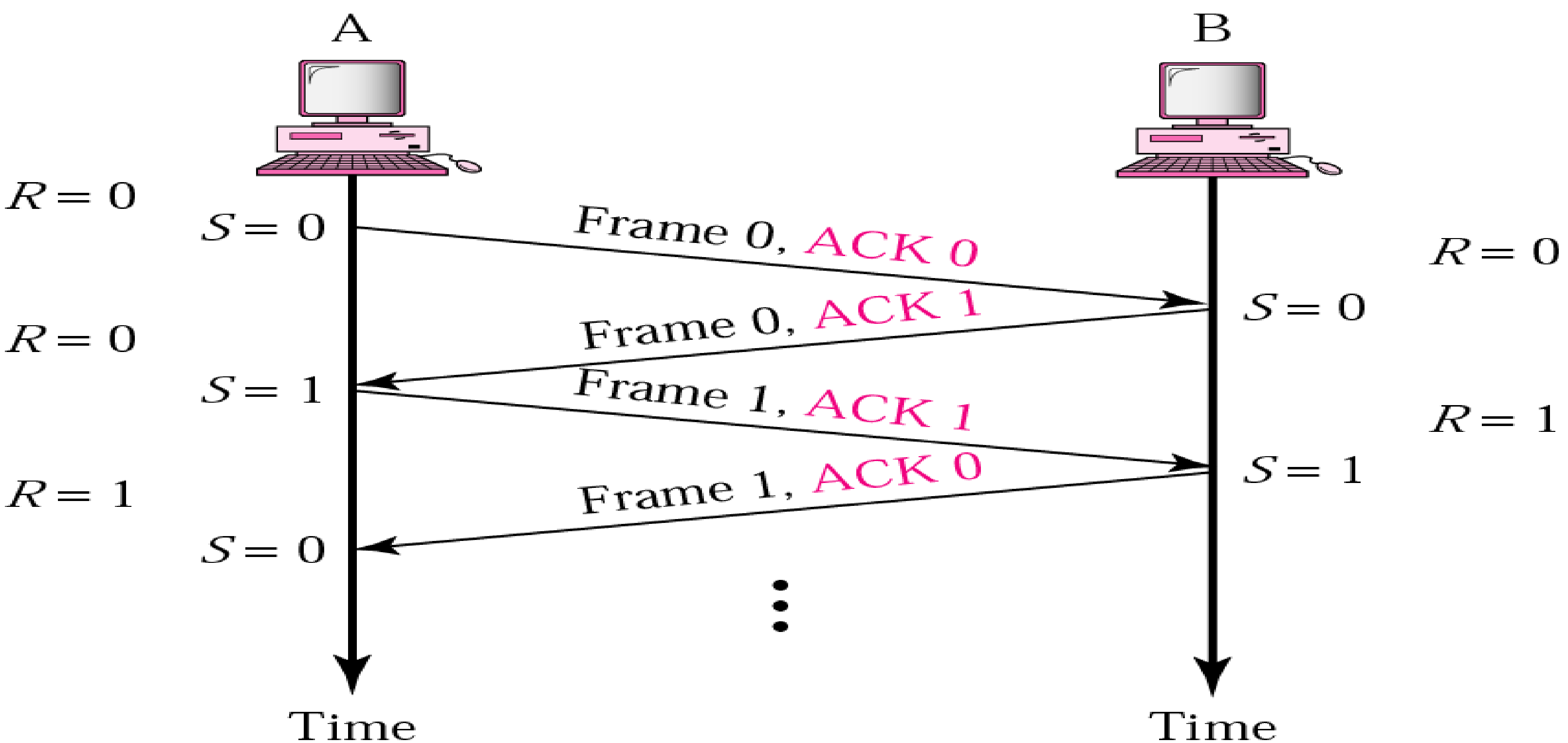
- Sending multiple types of packets in the same frame
- Example: Bidirectional channel
  - Receiver sends ACK with data
- Why
  - Saving on bandwidth
  - Saving of packet processing capacity of routers and switches
- Problem: What happens if there is no data to send?
- Use ad hoc mechanisms: e.g.
  -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

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

Piggybacking (Bidirectional transmission)

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

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



Pipelining

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

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

Sliding Window Protocols

1. One-Bit Sliding Window Protocol
2. Protocol Using Go Back N
3. Protocol Using Selective Repeat

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

Sliding window protocol

Sliding window protocols apply <u>Pipelining</u>

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

Sliding Window concept

Sender maintains window of frames it can send

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

Receiver maintains window of frames it can receive

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

Sliding Window Protocol

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

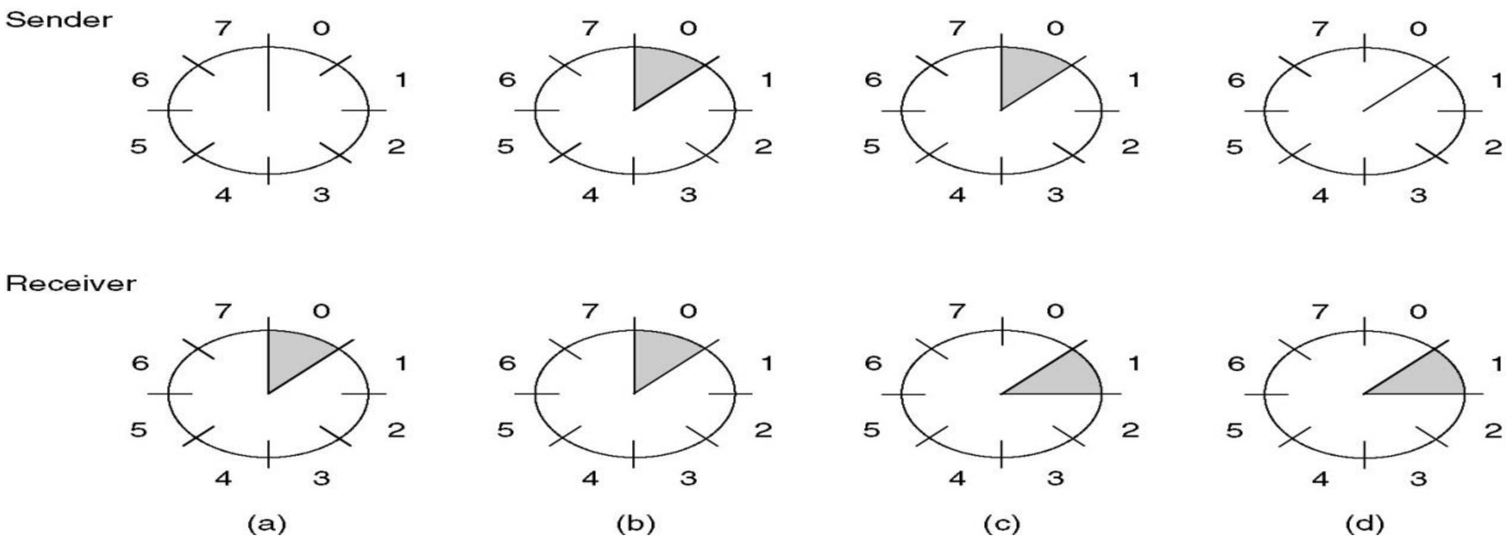
Sliding Window Protocol

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

Sliding Window Protocol

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

Sliding Window Protocols



- A sliding window of size 1, with a 3-bit sequence number.
  (a) Initially.
  (b) After the first frame has been sent.
  (c) After the first frame has been received.
  (d) After the first acknowledgement has been received.
- *Are packets delivered in order?*
  – Because the receiver window size is 1 packets, packets are always delivered in order

A One-Bit Sliding Window Protocol

```
/* Protocol 4 (sliding window) is bidirectional. */
#define MAX_SEQ 1                          /* must be 1 for protocol 4 */
typedef enum {frame_arrival, cksum_err, timeout} event_type;
#include "protocol.h"
void protocol4 (void)
{
  seq_nr next_frame_to_send;               /* 0 or 1 only */
  seq_nr frame_expected;                   /* 0 or 1 only */
  frame r, s;                              /* scratch variables */
  packet buffer;                           /* current packet being sent */
  event_type event;
  next_frame_to_send = 0;                  /* next frame on the outbound stream */
  frame_expected = 0;                      /* frame expected next */
  from_network_layer(&buffer);             /* fetch a packet from the network layer */
  s.info = buffer;                         /* prepare to send the initial frame */
  s.seq = next_frame_to_send;              /* insert sequence number into frame */
  s.ack =       frame_expected;            /* piggybacked ack */
  to_physical_layer(&s);                   /* transmit the frame */
  start_timer(s.seq);                      /* start the timer running */
```

Continued →

A One-Bit Sliding Window Protocol (ctd.)

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

```
while (true) {
        wait_for_event(&event);             /* frame_arrival, cksum_err, or timeout */
        if (event == frame_arrival) {       /* a frame has arrived undamaged. */
              from_physical_layer(&r);      /* go get it */

              if (r.seq == frame_expected) {     /* handle inbound frame stream. */
                    to_network_layer(&r.info);   /* pass packet to network layer */
                    inc(frame_expected);         /* invert seq number expected next */
              }

              If (r.ack == 1- next_frame_to_send) {   handle outbound frame stream. */
                    stop_timer(r.ack);               /* turn the timer off */
                    from_network_layer(&buffer);     /* fetch new pkt from network layer */
                    inc(next_frame_to_send);         /* invert senderís sequence number */
              }
        }
        s.info = buffer;                    /* construct outbound frame */
        s.seq = next_frame_to_send;         /* insert sequence number into it */
        s.ack =       frame_expected;       /* seq number of last received frame */
        to_physical_layer(&s);              /* transmit a frame */
        start_timer(s.seq);                 /* start the timer running */
   }
}
```

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

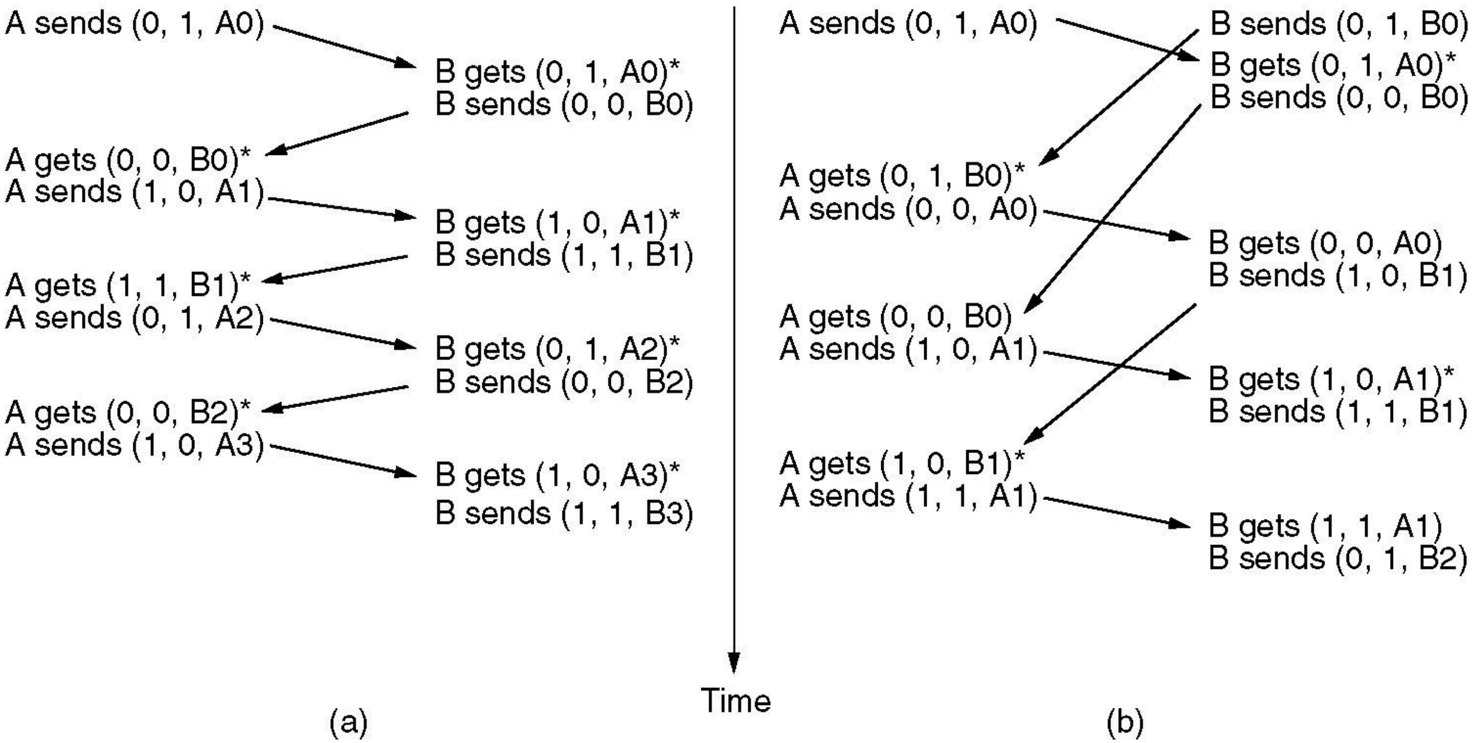
*This is equivalent to saying that ACK is the **circular**(frame expected-1)*
Circular(x-1) = (x>0) ? (x--) : MAX_SEQ

Assume calling start_timer() again restarts the timer

A One-Bit Sliding Window Protocol Correctness

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

A One-Bit Sliding Window Protocol (ctd.)



Two scenarios for protocol 4. (a) Normal case. (b) Abnormal case. The notation is *(seq, ack, packet number)*. An asterisk indicates where a network layer accepts a packet.

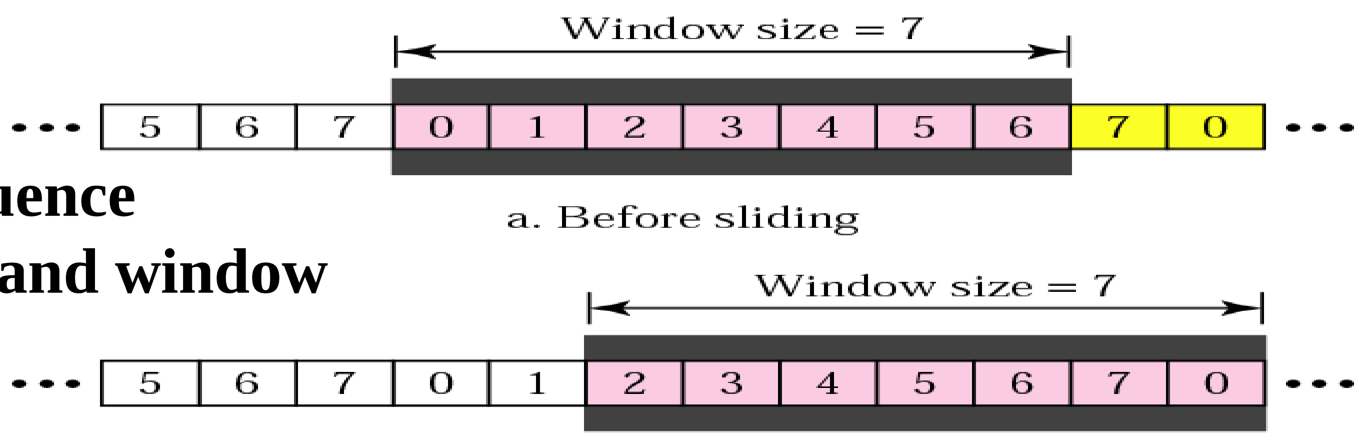
A One-Bit Sliding Window Protocol Disadvantages

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

Go_Back_N ARQ

Sender sliding window

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



a. Before sliding

b. After sliding two frames

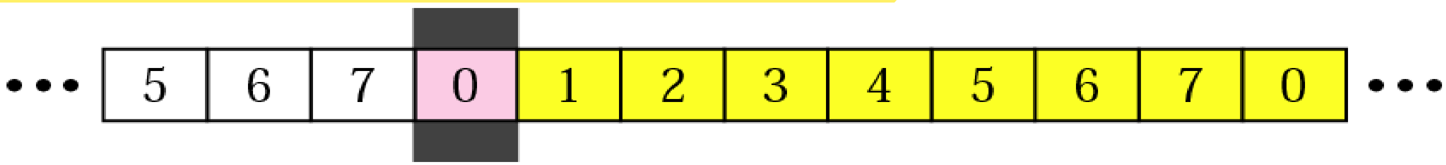
If m = 3; sequence numbers = 8 and window size =7

Acknowledged frames

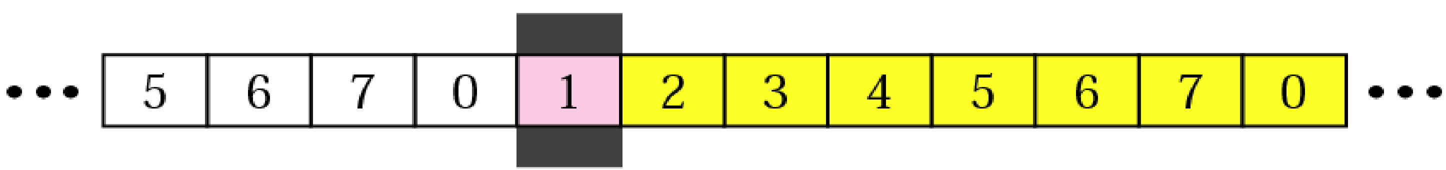
Go_Back_N ARQ

Receiver sliding window

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



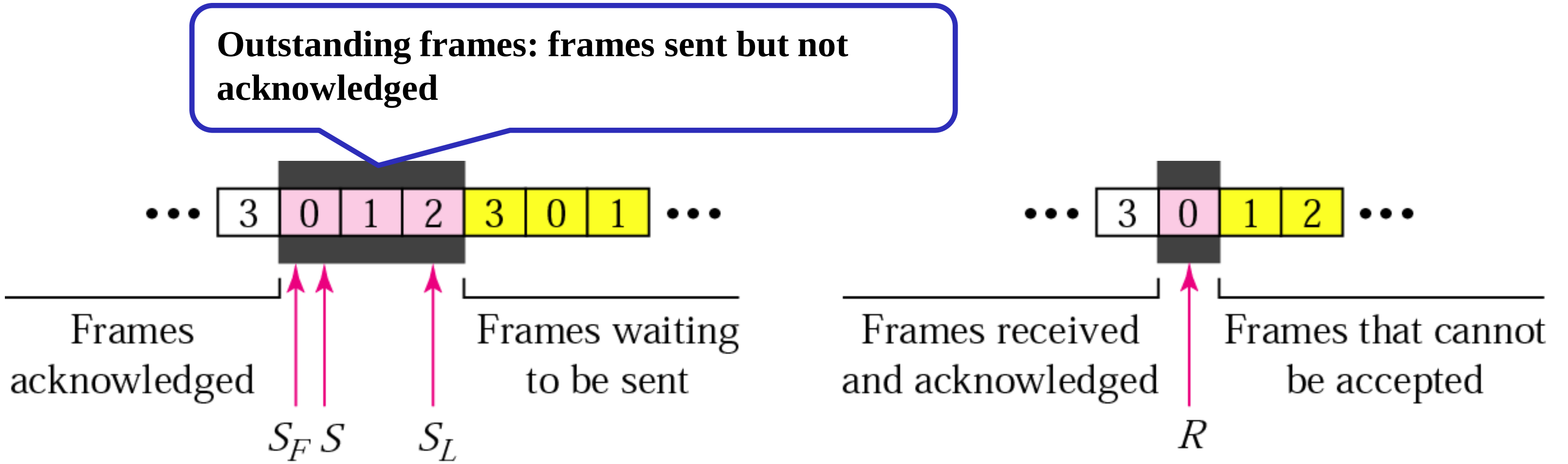
a. Before sliding



b. After sliding

Go-Back-N ARQ

control variables



a. Sender window

b. Receiver window

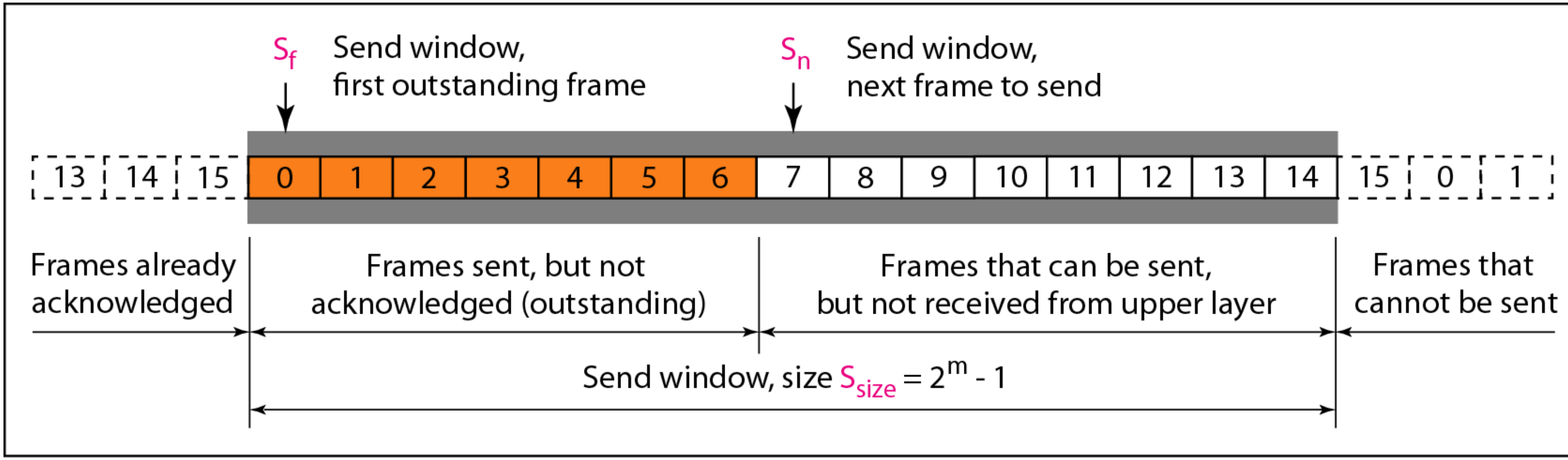
S: hold the sequence number of the recently sent frame

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

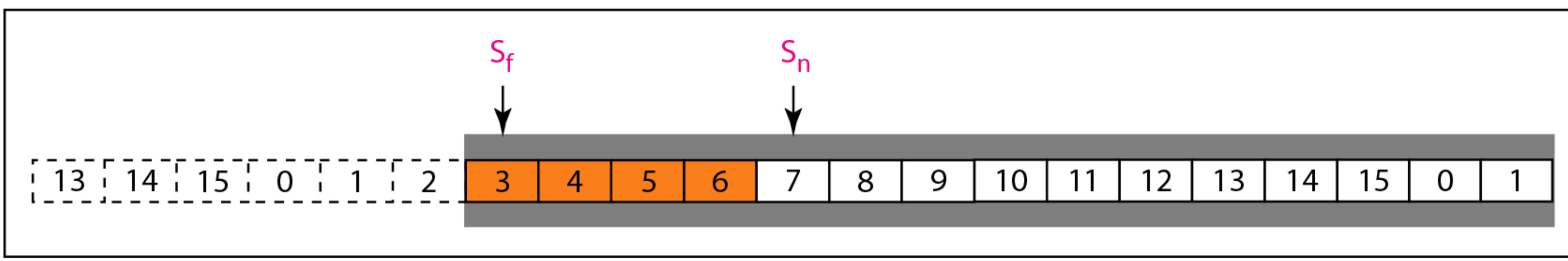
S_L: holds the sequence number of the last frame

R: sequence number of the frame expected to received

Go-Back-N ARQ



a. Send window before sliding



b. Send window after sliding

Go-Back-N ARQ

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

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

Go-Back-N ARQ

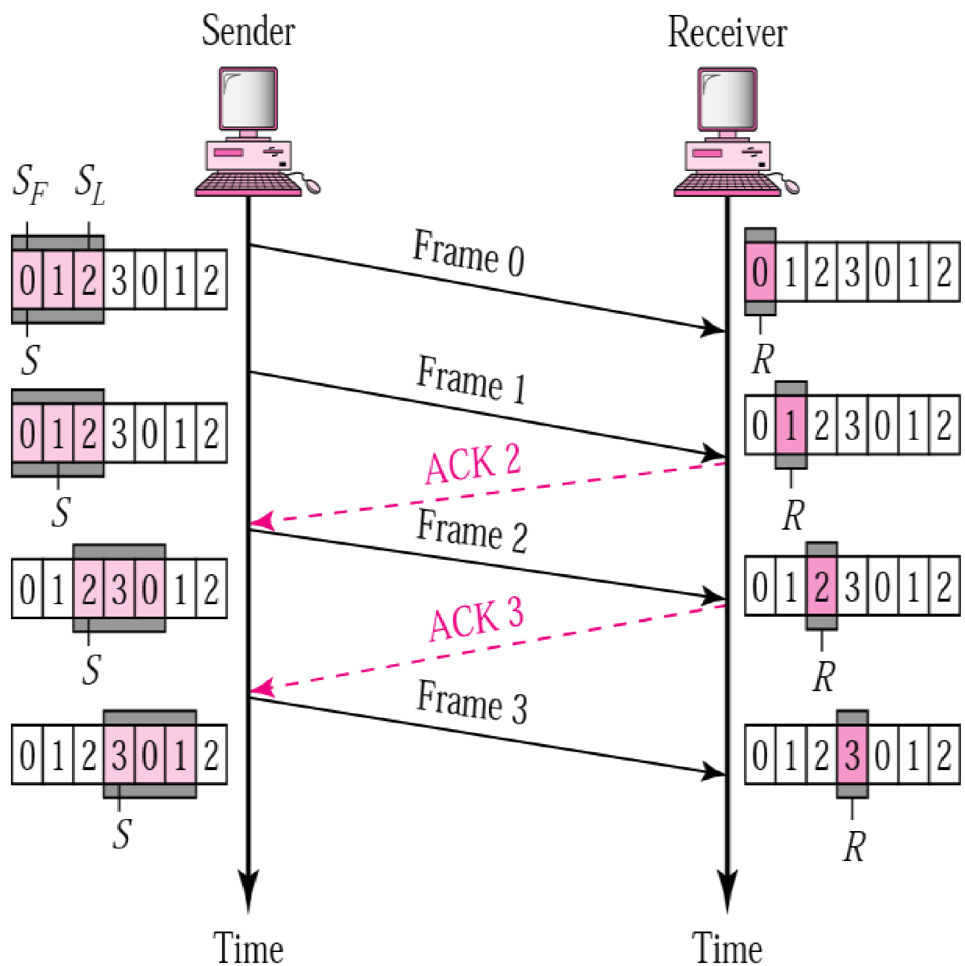
Example:

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

Go-Back-N ARQ

Normal operation

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

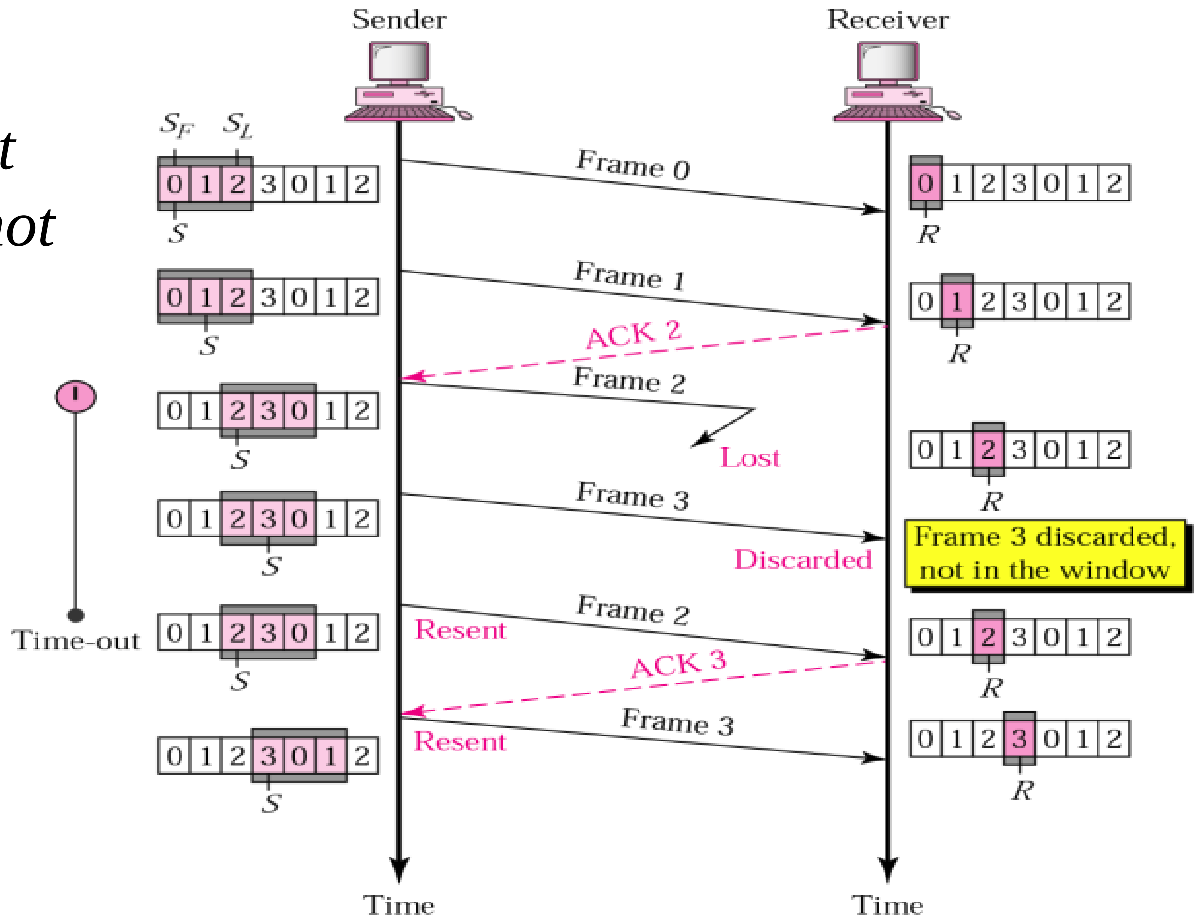


Go-Back-N ARQ

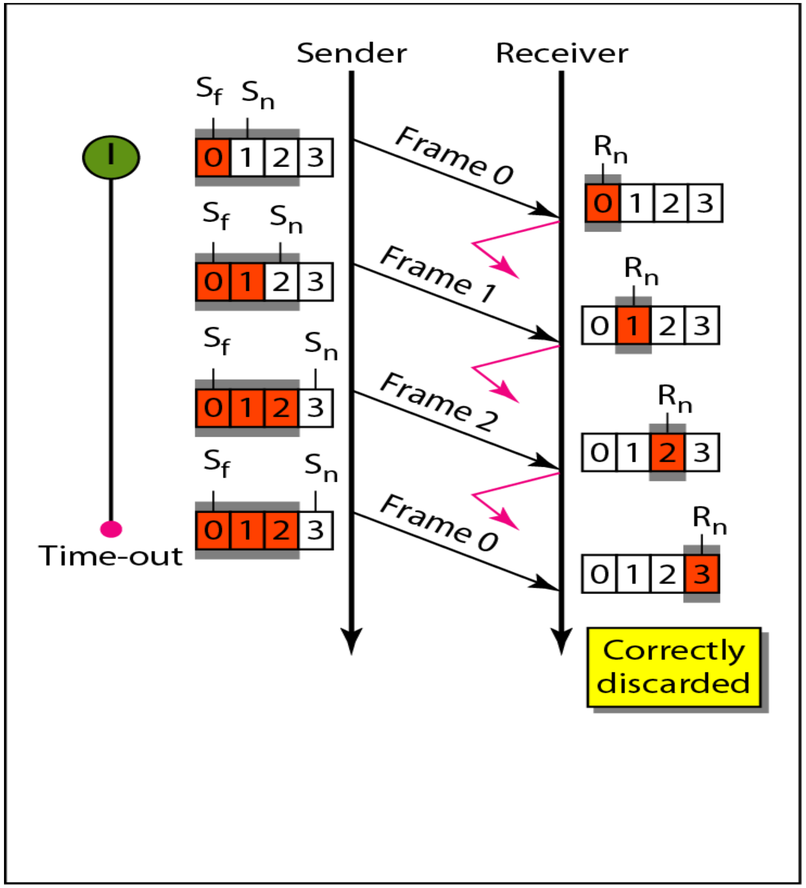
Damage or Lost Frame

Correctly received out of order packets are not Buffered

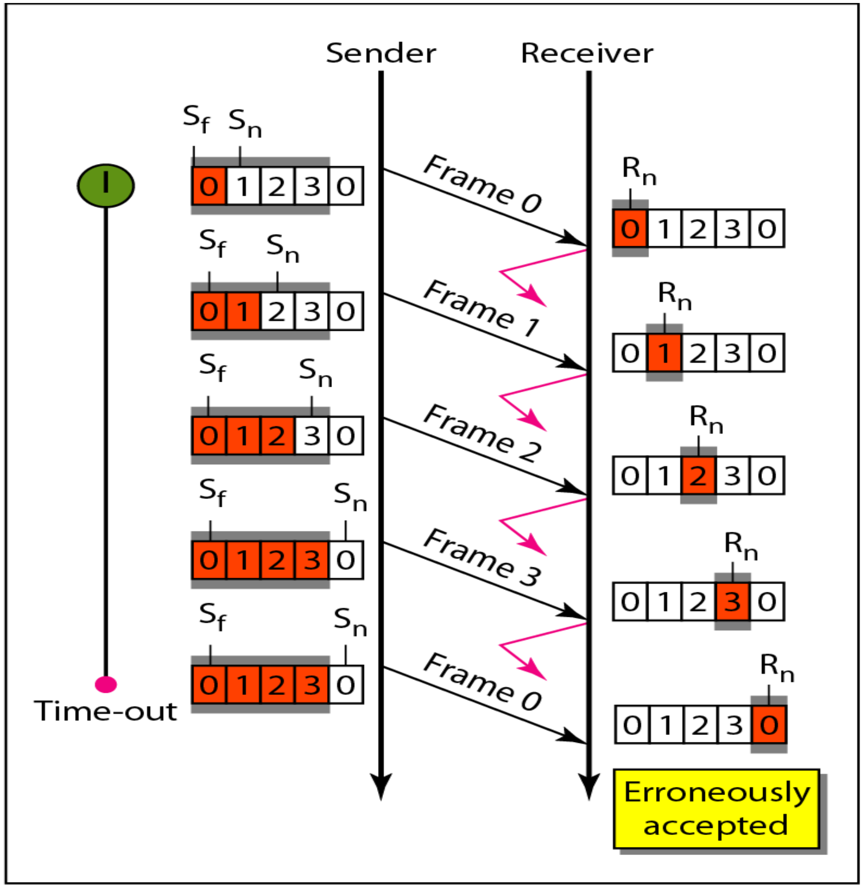
What is the disadvantage of this?



Go-Back-N ARQ



a. Window size < 2^m



b. Window size = 2^m

Go-Back-N ARQ

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

Bidirectional transmission : piggybacking

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

Note

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

Calculation of Window Size, Sequence number

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

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

Go Back N algorithm

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

Sliding Window Protocol Using Go Back N

```
/* Protocol 5 (pipelining) allows multiple outstanding frames. The sender may transmit up
  to MAX_SEQ frames without waiting for an ack. In addition, unlike the previous protocols,
  the network layer is not assumed to have a new packet all the time. Instead, the
  network layer causes a network_layer_ready event when there is a packet to send. */

#define MAX_SEQ 7                    /* should be 2^n – 1 */
typedef enum {frame_arrival, cksum_err, timeout, network_layer_ready} event_type;
#include "protocol.h"

static boolean between(seq_nr a, seq_nr b, seq_nr c)
{
/* Return true if a <=b < c circularly; false otherwise. */
  if (((a <= b) && (b < c)) || ((c < a) && (a <= b)) || ((b < c) && (c < a)))
      return(true);
    else
      return(false);
}
```

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

```
static void send_data(seq_nr frame_nr, seq_nr frame_expected, packet buffer[ ])
{
/* Construct and send a data frame. */
  frame s;                           /* scratch variable */

  s.info = buffer[frame_nr];         /* insert packet into frame */
  s.seq = frame_nr;                  /* insert sequence number into frame */
  s.ack = (frame_expected + MAX_SEQ) % (MAX_SEQ + 1);/* piggyback ack */
  to_physical_layer(&s);             /* transmit the frame */
  start_timer(frame_nr);             /* start the timer running */
}
```

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

- (Re-)Start a <u>separate</u> logical timer for every sent sequence number

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

Continued →

Sliding Window Protocol Using Go Back N

```
void protocol5(void)
{
  seq_nr next_frame_to_send;        /* MAX_SEQ > 1; used for outbound stream */
  seq_nr ack_expected;              /* oldest frame as yet unacknowledged */
  seq_nr frame_expected;            /* next frame expected on inbound stream */
  frame r;                          /* scratch variable */
  packet buffer[MAX_SEQ + 1];       /* buffers for the outbound stream */
  seq_nr nbuffered;                 /* # output buffers currently in use */
  seq_nr i;                         /* used to index into the buffer array */
  event_type event;

  enable_network_layer();           /* allow network_layer_ready events */
  ack_expected = 0;                 /* next ack expected inbound */
  next_frame_to_send = 0;           /* next frame going out */
  frame_expected = 0;               /* number of frame expected inbound */
  nbuffered = 0;                    /* initially no packets are buffered */
```

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

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

Continued →

Sliding Window Protocol Using Go Back N

```
while (true) {
  wait_for_event(&event);          /* four possibilities: see event_type above */

  switch(event) {
    case network_layer_ready:      /* the network layer has a packet to send */
         /* Accept, save, and transmit a new frame. */
         from_network_layer(&buffer[next_frame_to_send]); /* fetch new packet */
         nbuffered = nbuffered + 1;   /* expand the sender's window */
         send_data(next_frame_to_send, frame_expected, buffer);/* transmit the frame */
         inc(next_frame_to_send);     /* advance sender's upper window edge */
         break;

    case frame_arrival:            /* a data or control frame has arrived */
         from_physical_layer(&r);  /* get incoming frame from physical layer */

         if (r.seq == frame_expected) {
              /* Frames are accepted only in order. */
              to_network_layer(&r.info);  /* pass packet to network layer */
              inc(frame_expected);   /* advance lower edge of receiver's window */
         }
```

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

- Advance ***upper end*** of ***sender's*** window
$\Rightarrow$ increase sender's window

- Advance ***Lower end*** of ***receiver*** window
- *Remember* receiver window has **fixed size 1**
$\Rightarrow$ decrease receiver's window

Continued →

Sliding Window Protocol Using Go Back N

```
/* Ack n implies n – 1, n – 2, etc.  Check for this. */
while (between(ack_expected, r.ack, next_frame_to_send)) {
      /* Handle piggybacked ack. */
      nbuffered = nbuffered −1; /* one frame fewer buffered */
      stop_timer(ack_expected); /* frame arrived intact; stop timer */
      inc(ack_expected);       /* contract sender's window */
}
break;

case cksum_err: break;           /* just ignore bad frames */

case timeout:                    /* trouble; retransmit all outstanding frames */
   next_frame_to_send = ack_expected;   /* start retransmitting here */
   for (i = 1; i <= nbuffered; i++) {
      send_data(next_frame_to_send, frame_expected, buffer);/* resend 1 frame */
      inc(next_frame_to_send);   /* prepare to send the next one */
   }
}

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

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

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

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

⇒Reduce sender's window
⇒Free more buffers

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

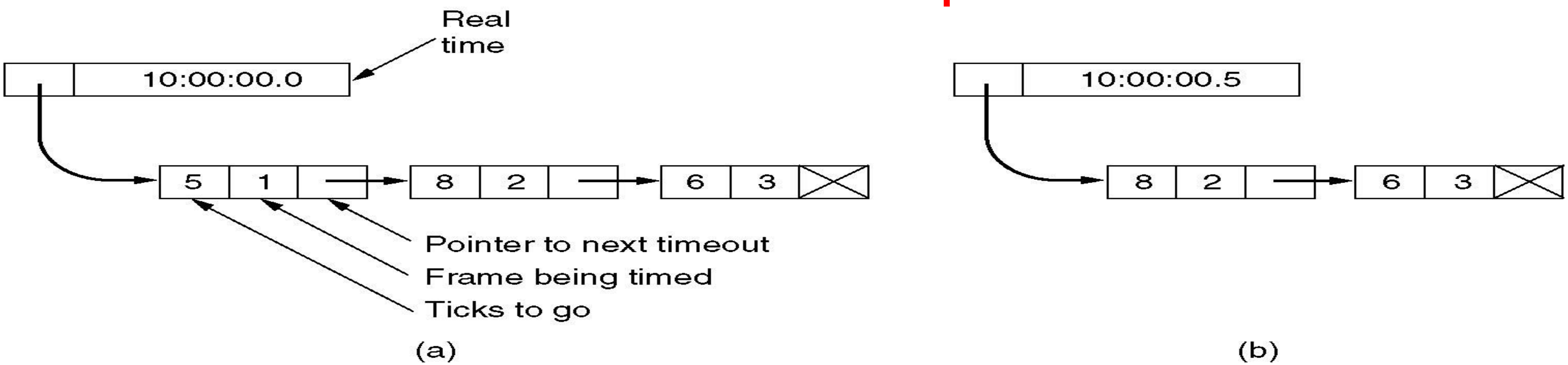
- Assume timeout because of frame loss
- Retransmit all frames ***starting*** from the ***last ACKed*** frame because receiver has window size one

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

Sliding Window Protocol Using Go Back N Discussion points

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

Sliding Window Protocol Using Go Back N
Efficient Timer Implementation



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

- Initially Three time outs are pending 10:00:05 , 10:00:13, 10:00:19
- Put sorted timers in a queue (linked list)
- If timer T2 expires after timer T1, the “Ticks-to-go” field in timer T2 has the number of ticks that timer T2 expire after timer T1
- Every clock tick, decrement the timer at the head of the queue
- When the queue head expires, fire the timer and delete the timer from the queue
- Minimum processing per clock tick
- *Stop_timer* causes scan of the queue to find the timer that needs to be removed
- *Start_timer* causes scan of the queue to find where to insert the started timer and to update “Ticks-to-go” for timers ***after*** the inserted timers
- Argument of *start_timer* and *stop_timer* indicates which timer to start or stop

To be continued …