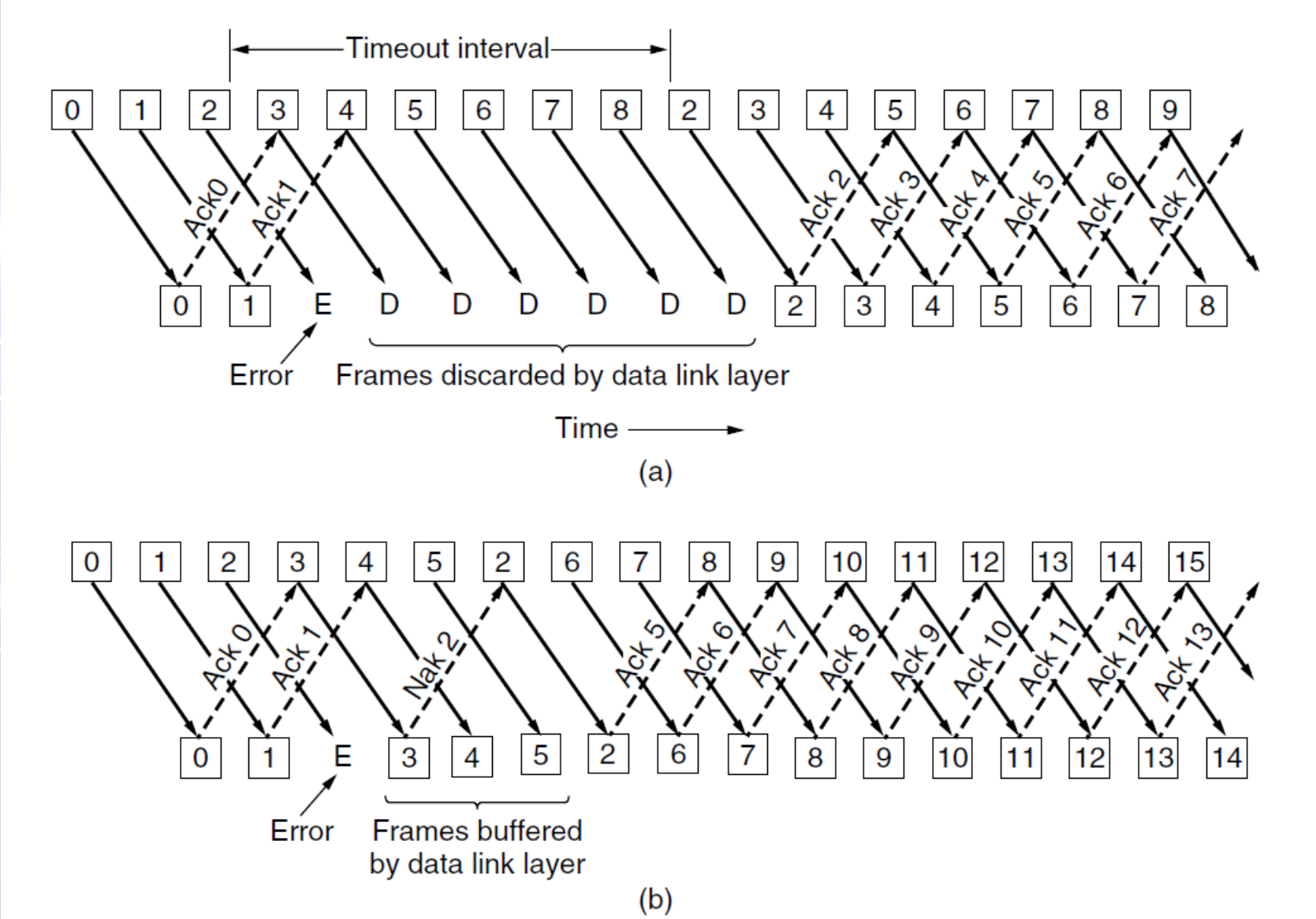
# Go Back N vs. Selective Repeat




Pipelining and error recovery. Effect of an error when
(a) Go back N: receiver's window size is 1 and (b)Selective Repeat: receiver's window size is large.

# Go Back N vs. Selective Repeat

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

*→Tradeoff between buffer space and link bandwidth utilization*

# Go Back N

- 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
- *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

# Go Back N

- Circular sequence number:

```
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 earlier than ***c***
- Checks if ***b*** is within the window

| | | | | | | |
|---|---|---|---|---|---|---|
| a=0 | 1 | 2 | b=3 | 4 | 5 | c=6 |
| a=4 | 5 | 6 | b=7 | 0 | 1 | c=2 |
| a=7 | 0 | 1 | b=2 | 3 | 4 | c=5 |

# Go Back N

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

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

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

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

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

- **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.

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

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

# 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 */
```

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

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

# 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
- We disable network layer when sender window is full
- Hence, we will always attempt to transmit the entire sender window

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

⇒ increase sender's window

# 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();
  }
}
```

e.g. if Ack 3
then packets 2,1,0,…,
arrived.
called:
**cumulative acknowledgement**

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

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

⇒Reduce sender's window
⇒Free more buffers

- 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)
- Maximum number of outstanding packets is MAX_SEQ ***NOT*** (MAX_SEQ+1)
  - Because the network layer is enabled only if nbuffered is strictly less than MAX_SEQ
- There are at most MAX_SEQ+1 distinct sequence numbers

# Go Back N

- If there is **no reverse traffic**, protocol **fails**
  - sender will not receive ACK and not advance the window and it will keep timing-out and retransmitting → deadlock.
- Too many errors result in poor throughput because of retransmission.
- **No buffering** at the receiver since the receiver window size is 1.
- There is **buffering** at the sender since on timeout the sender re-transmits all outstanding frames.
- Need for *logically separate timer* per outstanding frame.
  - Expedite retransmission on loss (Optimization):
    - Sender sends packets 0→6 and puts timer on 6 (last set timer).
    - Suppose 0 is lost , 0 will be retransmitted only after timer of 6 expires.
  - Failure: (This can only happen if the protocol implementation is explicitly changed to allow: "setting the timer every n packets only")
    - Suppose sender puts timer every 7 packets and that the NL has only 5 packets to send.
    - If any of the 5 packets gets lost, no retransmission since no timeout.

# Go Back N

Sender window size (maximum outstanding frames) ***must be*** MAX_SEQ and ***not*** MAX_SEQ+1. I.e. if MAX_SEQ=7, we have 8 distinct sequence numbers (0→7) but sender window size=7(***Why??***)

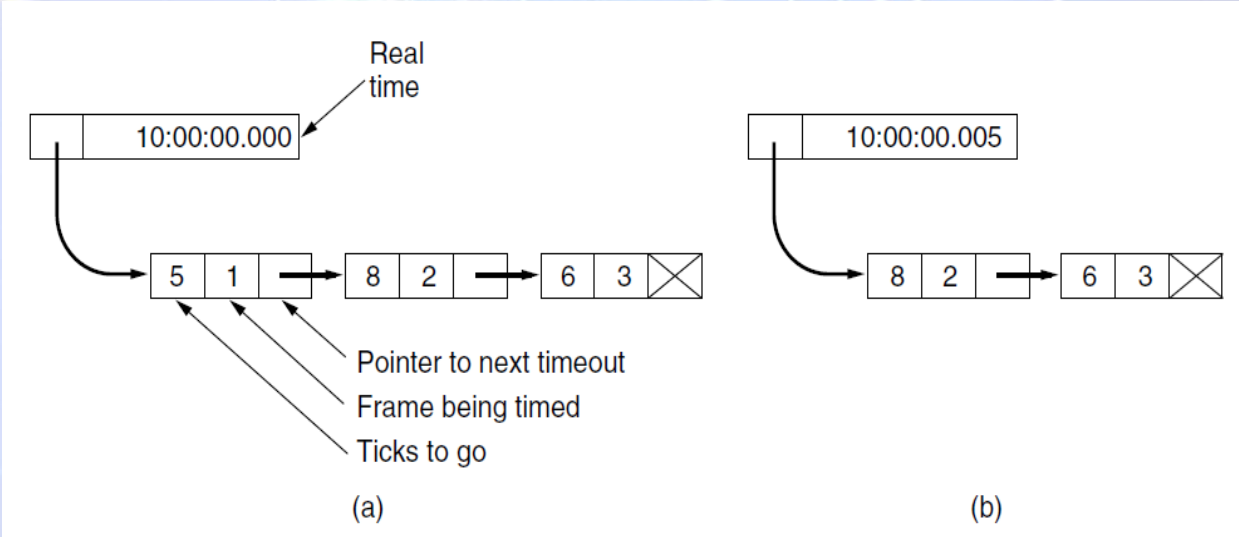
- Assume sender window size=MAX_SEQ+1
- A transmits 0,1,2,3,4,5,6,7
- B received all of them and send ACK 7
- A sends another batch 0→7
- All 8 packets in the second batch are lost
- Now B has some data
- B sends data to A with ACK=7
- It is impossible for A to know whether the ACK is for the second batch or a duplicate ACK for the first batch

**In the correct protocol case:**

- A sends 0→6
- B sends ACK 6
- A sends a second batch 7,0,1,2,3,4,5
- Second batch is lost
- B sends data with ACK=6
- 6 is outside the sending window of A→ A will ignore.

# Go Back N: Efficient Timer Implementation

- Each frame times out independently of all the other ones. However, all of these timers can easily be simulated in software using a single hardware clock that causes interrupts periodically.
- The pending timeouts form a linked list, with each node of the list containing
  - the number of clock ticks until the timer expires.
  - the frame being timed.
  - a pointer to the next node.




(a) The queued timeouts.(b) The situation after the first timeout has expired.

- Initially, the real time is 10:00:00.000: three timeouts are pending, at 10:00:00.005, 10:00:00.013, and 10:00:00.019.
- Every time the hardware clock ticks, the real time is updated and the tick counter at the head of the list is decremented.
- When the tick counter becomes zero, a timeout is caused and the node is removed from the list.
- This organization requires the list to be scanned when *start timer* or *stop timer* is called, but it does not require much work per tick.