Chapter 3- Part 5

The Data Link Layer

Sliding Windows Protocols (2)

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

Selective Repeat _ARQ Algorithms

Selective Repeat ARQ

*Go-Back-N ARQ is inefficient in the case of a **noisy** link.*

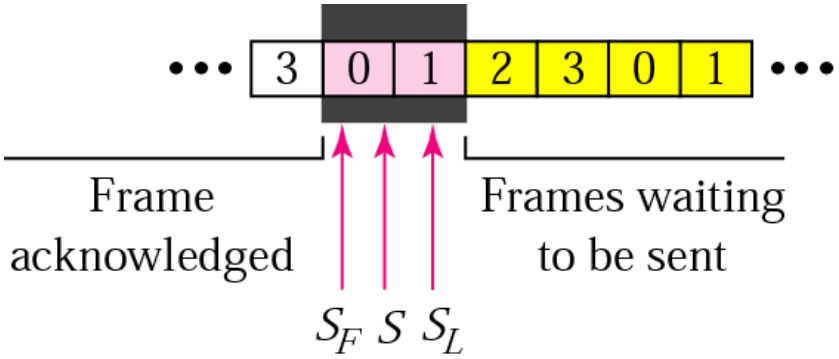
- *In a noisy link frames have higher probability of damage , which means the resending of multiple frames.*
- *this resending consumes the bandwidth and slow down the transmission .*

***Solution**:*

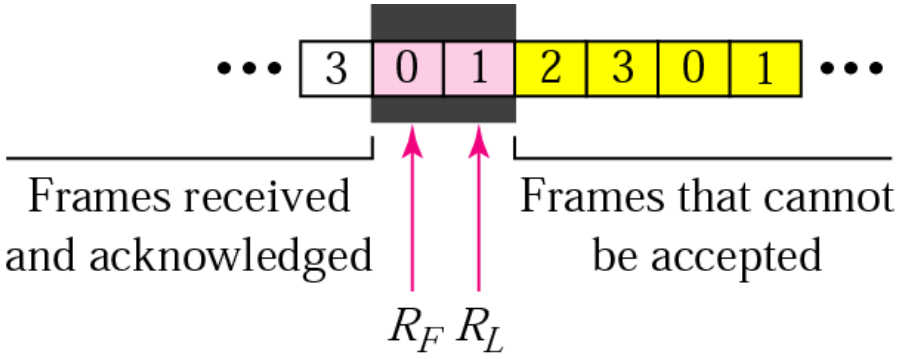
- *Selective Repeat ARQ protocol : resent only the damage frame*
- *It defines a **negative Acknolgment (NAK)** that report the sequence number of a damaged frame before the timer expires*
- *It is more efficient for noisy link, but the processing at the receiver is more complex*

Selective Repeat ARQ

- *The window size is reduced to one half of* 2^m
- *Sender window size = receiver window size =* 2^m */2*
- *Window size =* **(Max sequence number +1) /2**
- *If m = 2, Window size = 4/2=2*
- *Sequence number = 0, 1, 2 , 3*



a. Sender window

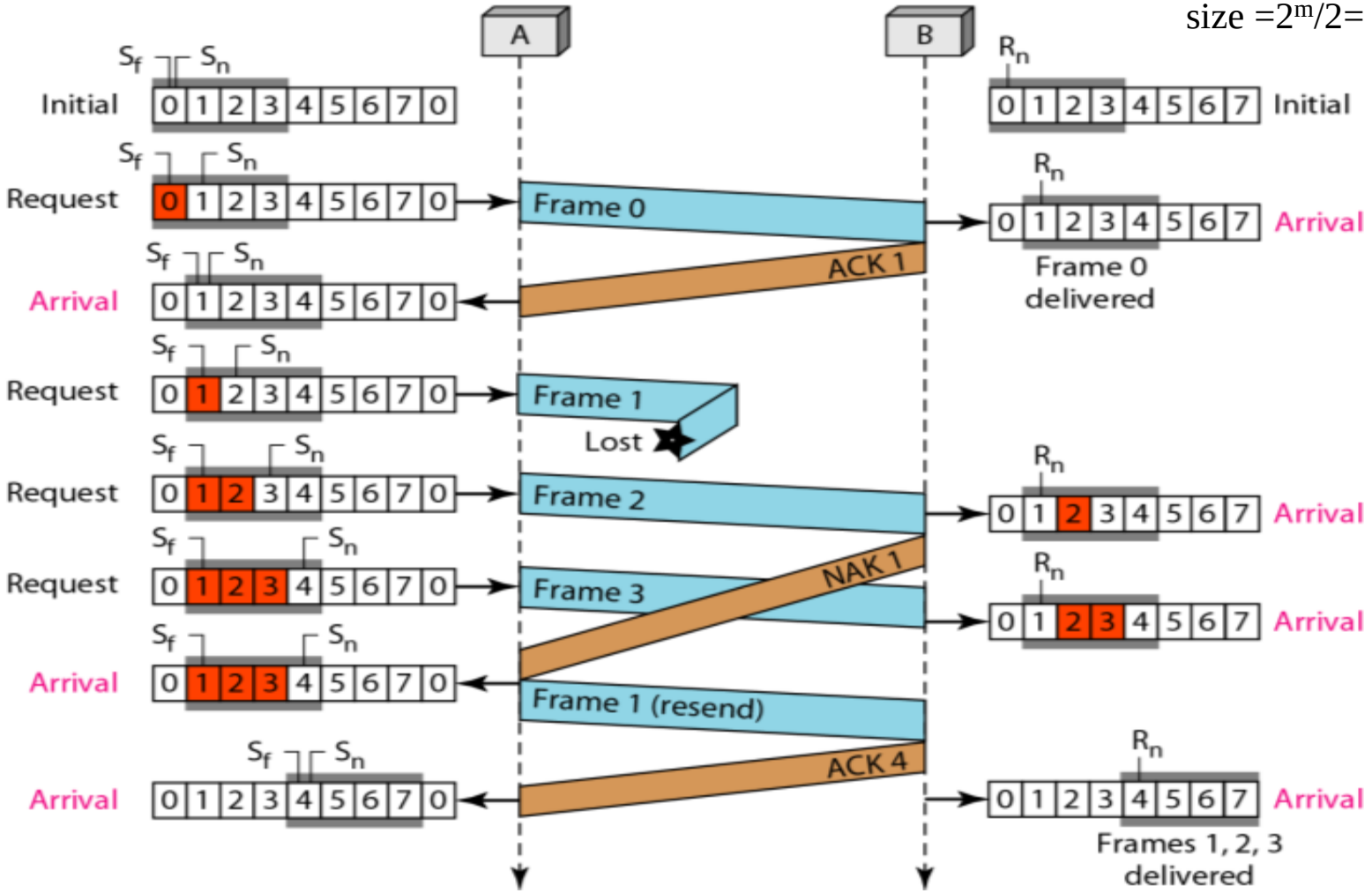


b. Receiver window

Selective Repeat ARQ

m=3
Sequences no=2^m =8 :
0,1,2 ,3,4,5,6,7Window
size =2^m/2= 8/2=4

Lost Frame



Selective Repeat ARQ

- *At the receiver site we need to distinguish between the acceptance of a frame and its delivery to the network layer.*
- *At the second arrival , frame 2 arrives and is stored and marked , but it can not be delivered because frame 1 is missing .*
- *At the next arrival , frame 3 arrives and is marked and stored , but still none of the frames can be delivered .*
- *Only at the last arrival , when finally a copy of frame 1 arrives , can frames 1 , 2 , and 3 be delivered to the network layer.*
- ***There are two conditions for the delivery of frames to the network layer: First , a set of consecutive frames must have arrived. Second, the set starts from the beginning of the window .***

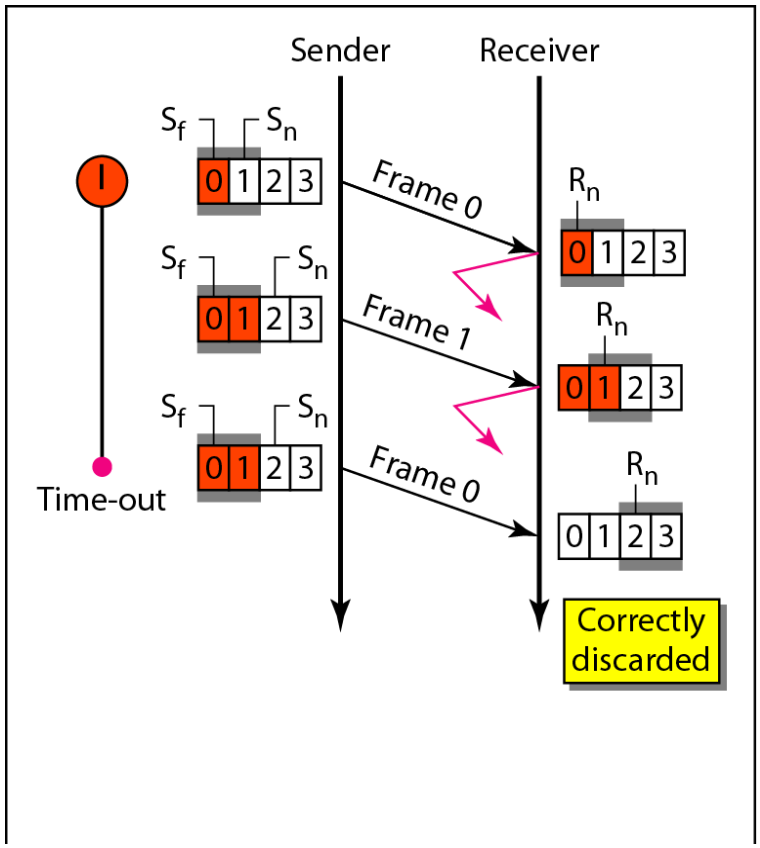
Selective Repeat ARQ

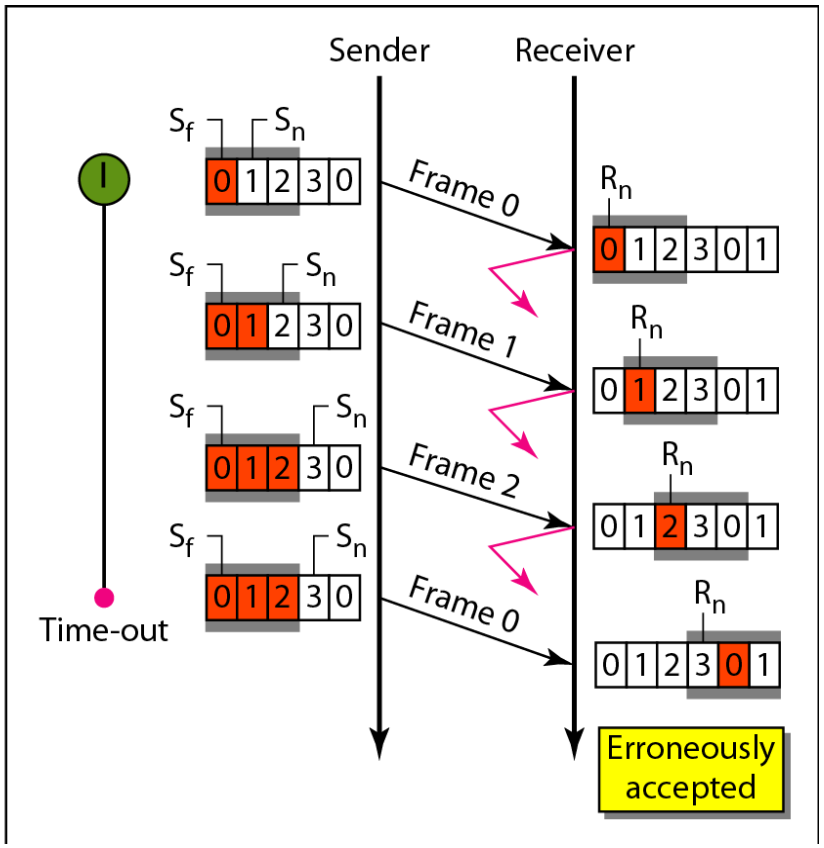
The next point is about the ACKs:

- *Notice that only two ACKs are sent here. The first one acknowledges only the first frame; the second one acknowledges three frames.*
- *In Selective Repeat, ACKs are sent when data are delivered to the network layer. If the data belonging to n frames are delivered in one shot , only one ACK is sent for all of them.*

Selective Repeat ARQ

m=2, Window size =2 , seq numbers is 4 (0,1,2,3)





Note

In Selective Repeat ARQ, the size of In the sender and receiver window must be at most one-half of 2^m.

SR_ARQ Algorithms

Selective Repeat Algorithm

- Accepts ***out of order*** frames
- **One timer per frame**
- On timer expire, only the frame associated with timer is re-sent ⇒ *selective repeat*
- Receiver window size is **fixed** at (MAX_SEQ+1)/2
- Sender window size starts at 0 and *grows* to (MAX_SEQ+1)/2
- We have timer for ACK ⇒ No need for reverse traffic to piggyback ACK
- For each sequence number within window, receiver has
  - a buffer
  - "arrived" bit. If set, means buffer is full
- When a packet arrives at receiver
  - Check to see if it falls within window of sequence numbers
  - Check to see if arrive bit is clear (i.e. buffer is empty)
- If both conditions are satisfied, store the arrived packet
- Receiver delivers packets to network layer in order and advances window
- Assume that the function *start_ack_timer()* **resets** the expiration time of the ACK timer to the current time.

A Sliding Window Protocol Using Selective Repeat

Sender & Receiver window size is **half** of MAX_SEQ

```
/* Protocol 6 (nonsequential receive) accepts frames out of order, but passes packets to the
   network layer in order. Associated with each outstanding frame is a timer. When the timer
   expires, only that frame is retransmitted, not all the outstanding frames, as in protocol 5. */

#define MAX_SEQ 7                                   /* should be 2^n – 1 */
#define NR_BUFS ((MAX_SEQ + 1)/2)
typedef enum {frame_arrival, cksum_err, timeout, network_layer_ready, ack_timeout} event_type;
#include "protocol.h"
boolean no_nak = true;                              /* no nak has been sent yet */
seq_nr oldest_frame = MAX- SEQ + 1;                 /* initial value is only for the simulator */

static boolean between(seq_nr a, seq_nr b, seq_nr c)
{
/* Same as between in protocol5, but shorter and more obscure. */
  return ((a <= b) && (b < c)) || ((c < a) && (a <= b)) || ((b < c) && (c < a));
}

static void send_frame(frame_kind fk, seq_nr frame_nr, seq_nr frame_expected, packet buffer[ ])
{
/* Construct and send a data, ack, or nak frame. */
  frame s;                                          /* scratch variable */

  s.kind = fk;                                      /* kind == data, ack, or nak */
  if (fk == data) s.info = buffer[frame_nr % NR_BUFS];
  s.seq = frame_nr;                                 /* only meaningful for data frames */
  s.ack = (frame_expected + MAX_SEQ) % (MAX_SEQ + 1);
  if (fk == nak) no_nak = false;                    /* one nak per frame, please */
  to_physical_layer(&s);                            /* transmit the frame */
  if (fk == data) start_timer(frame_nr % NR_BUFS);
  stop_ack_timer();                                 /* no need for separate ack frame */
}
```

If frame type is ***not*** *data*, **s.seq** is ***ignored***
⇒ We can put anything

- ***s.ack*** contains the sequence number of the ***last received*** frame in the ***contiguous*** sequence of frames
- you can think of ***s.ack*** as **circular(frame_expected-1, MAX_SEQ)**

If the packet that we are sending now is a NAK, then clear **no_nak** This way if the next packet is NOT **frame_expected,** we will start the **ACK timer** instead of sending a NAK one more time

Continued →

A Sliding Window Protocol Using Selective Repeat (2)

```
void protocol6(void)
{
  seq_nr ack_expected;                  /* lower edge of sender's window */
  seq_nr next_frame_to_send;            /* upper edge of sender's window + 1 */
  seq_nr frame_expected;                /* lower edge of receiver's window */
  seq_nr too_far;                       /* upper edge of receiver's window + 1 */
  int i;                                /* index into buffer pool */
  frame r;                              /* scratch variable */
  packet out_buf[NR_BUFS];              /* buffers for the outbound stream */
  packet in_buf[NR_BUFS];               /* buffers for the inbound stream */
  boolean arrived[NR_BUFS];             /* inbound bit map */
  seq_nr nbuffered;                     /* how many output buffers currently used */
  event_type event;

  enable_network_layer();               /* initialize */
  ack_expected = 0;                     /* next ack expected on the inbound stream */
  next_frame_to_send = 0;               /* number of next outgoing frame */
  frame_expected = 0;
  too_far = NR_BUFS;
  nbuffered = 0;                        /* initially no packets are buffered */
  for (i = 0; i < NR_BUFS; i++) arrived[i] = false;
```

Need buffer at ***both*** sender and receiver

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

receiver window is **fixed** size NR_BUFS. It is *initialized* between **0** and **NR_BUFs**

Continued →

A Sliding Window Protocol Using Selective Repeat (3)

out_buf[] is ***circular***

```
while (true) {
  wait_for_event(&event);                    /* five possibilities: see event_type above */
  switch(event) {
    case network_layer_ready:                /* accept, save, and transmit a new frame */
         nbuffered = nbuffered + 1;          /* expand the window */
         from_network_layer(&out_buf[next_frame_to_send % NR_BUFS]); /* fetch new packet */
         send_frame(data, next_frame_to_send, frame_expected, out_buf);/* transmit the frame */
         inc(next_frame_to_send);            /* advance upper window edge */
         break;

    case frame_arrival:                      /* a data or control frame has arrived */
         from_physical_layer(&r);            /* fetch incoming frame from physical layer */
         if (r.kind == data) {
              /* An undamaged frame has arrived. */
              if ((r.seq != frame_expected) && no_nak)
                   send_frame(nak, 0, frame_expected, out_buf); else start_ack_timer();
              if (between(frame_expected, r.seq, too_far) && (arrived[r.seq%NR_BUFS] == false)) {
                   /* Frames may be accepted in any order. */
                   arrived[r.seq % NR_BUFS] = true;      /* mark buffer as full */
                   in_buf[r.seq % NR_BUFS] = r.info;     /* insert data into buffer */
                   while (arrived[frame_expected % NR_BUFS]) {
                        /* Pass frames and advance window. */
                        to_network_layer(&in_buf[frame_expected % NR_BUFS]);
                        no_nak = true;
                        arrived[frame_expected % NR_BUFS] = false;
                        inc(frame_expected);     /* advance lower edge of receiver's window */
                        inc(too_far);            /* advance upper edge of receiver's window */
                        start_ack_timer();       /* to see if a separate ack is needed */
                   }
              }
         }
```

- We did NOT receive the lower end of receive window
⇒Send NAK immediately
- Because ***send_frame()*** sets ***s.ack*** to **circular(frame_expected-1)** then the NAK frame contains the sequence number of the last received frame
- Thus a when a sender receives a NAK, it should re-send the frame with sequence number **circular(r.ack+1)**

Because we are sending a NAK, this value is meaningless ⇒ Put any thing

Restart ACK timer every time we receive out of order packet (***See WHY later?***)

Deliver ***contiguous*** packet sequence to network layer starting from **bottom** of receiver window. ***frame_expected*** is ***one more than*** the sequence number of the **last received frame** in ***contiguous*** sequence

- Increment **both** lower and upper receiver window boundaries
⇒ Receiver window size *always* fixed at **NR_BUFs**

Allow sending NAK because we send NAK for ***frame_expected*** and ***frame_expected*** will be incremented soon

Reset ACK timer to the current time

A Sliding Window Protocol Using Selective Repeat (4)

- When the other side sends a NAK, the other side sets the **"ack"** field to **circular (frame_expected – 1).**
- Hence **r.ack** contains sequence number of the last received frame by the other side.
- Thus a sender should re-send the frame whose sequence number is **circular(r.ack+1)**
- Thus we have to test ***between*** for **circular(r.ack+1)**

- ACK means all frames in the **contiguous** sequence of frames **before and including** ACKed frame have been received. Do the following *between* **ack_expected** and **r.ack**

⇒Free buffers
⇒Stop all retransmit timers
⇒Advance sender **lower** window

```
if((r.kind==nak) && between(ack_expected,(r.ack+1)%(MAX_SEQ+1),next frame to send))
        send_frame(data, (r.ack+1) % (MAX_SEQ + 1), frame_expected, out_buf);

while (between(ack_expected, r.ack, next_frame_to_send)) {
        nbuffered = nbuffered –1;           /* handle piggybacked ack */
        stop_timer(ack_expected % NR_BUFS);     /* frame arrived intact */
        inc(ack_expected);                  /* advance lower edge of sender's window */
        }
        break;
case cksum_err:
        if (no_nak) send_frame(nak, 0, frame_expected, out_buf);/* damaged frame */
        break;
case timeout:
        send_frame(data, oldest_frame, frame_expected, out_buf);/* we timed out */
        break;
case ack_timeout:
        send_frame(ack,0,frame_expected, out_buf);      /* ack timer expired; send ack */
    }
    if (nbuffered < NR_BUFS) enable_network_layer(); else disable_network_layer();
  }
}
```

We assume that the timeout event causes the variable ***oldest_frame*** to be set according to the timeout that just expired

Re-send NAK for last frame received if we get a corrupted frame

- On ACK timeout, we send ACK for the frame **before** bottom of the receiver window because **frame_expected** is one more than the sequence number of the last frame received in **contiguous** sequence
- Because *send_frame()* sets ***s.ack*** to **circular(frame_exptected-1)**, we will end up *re-ACKing* the last frame received in the **contiguous** sequence

Because we have ACK timeout
⇒No need for reverse traffic
⇒Solves the blocking problem of Go back N

Window size is NR_BUF
⇒ half of MAX_SEQ

A Sliding Window Protocol Using Selective Repeat Discussion Points

- Why is NR_BUFS defined (MAX_SEQ+1)/2?
- How many buffers must receiver have?
- How may timers are needed at the receiver?
- How to get away without reverse traffic?
- How is NAK making protocol more efficient
- How to adjust timers
- ***Why do we have to restart ACK timer every time we receive out of order packet?***

A Sliding Window Protocol Using Selective Repeat Why NR_BUFS (MAX_SEQ+1)/2

- Suppose we allow NR_BUFS to be MAX_SEQ-1 (just like Go Back N)
- Consider the following sequence of events
- Sender window is 0,1,2,3,4,5,6
  - `next_frame_to_send` = 7, `ack_expected` = 0
- Receiver window is 0,1,2,3,4,5,6
- Sender sends frames 0,1,2,3,4,5,6
- Receiver receives all seven frames and sends ACK for all of them
  - All frames are valid frames because their sequence numbers lie within receiver window
  - Receiver delivers all 7 frames to network layer and sends ACK for 0,1,2,3,4,5,6
  - Receiver advances its receive window to 7,0,1,2,3,4,5
    - $\Rightarrow$ `Frame_expected` = 7, `too_far` = 6
- Suppose all ACKs are lost
- Sender times out. Thus it **resends** frame 0 because frame 0 timer is the first timer started and hence it will be the first timer to expire
- Receiver receives **resent** data frame with s.seq = 0
  - S.ack = 0 is within the new receiver window
  - Receiver accepts frame 0 $\Rightarrow$ **DUPLICATE** packet because this packet is already delivered to network layer
  - arrived[0] = TRUE
  - Does NOT advance `frame_expected` because `frame_expected` not received yet

A Sliding Window Protocol Using Selective Repeat Why NR_BUFS (MAX_SEQ+1)/2

- Now receiver ACK timeout expires
  - Resends ACK with s.ack = circular(`frame_expected` – 1) = 6
- Sender receives ACK with s.ack = 6
  - Sender assumes that all frames 0,1,2,3,4,5,6 have been received
  - Sender advances its window to so that the sender window = 7,0,1,2,3,4,5
  - Sender sends frames 7,0,1,2,3,4,5

  ⇒ `ack_expected` = 7, `next_frame_to_send` = 6
- Receiver receives frame 7,0,1,2,3,4,5
  - Receiver **accepts** frame 7 and puts it in buffer
  - Receiver **rejects** frame 0 because arrived[0]=TRUE ⇒ The new frame 0 is **LOST**
  - Receiver **Accepts** 1,2,3,4,5 and puts them in buffer
  - Now receiver has a contiguous sequence of frames in the receive window so it delivers 7,0,1,2,3,4,5 to network layer
  - The delivered packets have the following problems
    - Frame 0 is **duplicate** and **out of sequence** because it belongs to the first batch
    - Frame 0 from the second batch is **lost**
- NOTE:
  - When Receiver receives frame 0 for the second time, it will send a NAK for frame_expected. Thus s.ack=6 in the NAK
  - The sender receives a NAK with r.ack = 6
  - The sender checks if **circular(r.ack+1)** lies between **ack_expected** and **next_frame_to_send** before resending the NAKed frame
    - circular(r.ack+1) = 7, ack_expected = 0, and next_frame_to_send = 7
    - Thus the check **Fails**
    - Hence the NAK will have **no effect**

A Sliding Window Protocol Using Selective Repeat How many buffers and timers at receiver

- Receiver only accepts frames within the window size
- Window size is **half** the sequence number*
- Receiver needs buffers equal to window size **not** MAX_SEQ
- We need one timer ONLY at receiver: This is the ACK timer
- We need timers equal to window size, not MAX_SEQ at the sender

A Sliding Window Protocol Using Selective Repeat Auxiliary Timer and NAK

- Auxiliary ACK timer
- No need for reverse traffic
- Auxiliary ACK timer should be *shorter* than retransmit timer
- NAK to expedite retransmit (not needed for correctness)
- Send NAK on
  - Out of sequence: `r.seq ≠ frame_expected`
  - Checksum error: possible damage
  - In both cases, we send NAK for **circular(frame_expected – 1)**
  - Remember that *circular(x-1) = (x>0) ? (x--) : MAX_SEQ*
- How to avoid multiple NAKs per packet
  - Variable `no_nak` is set only if receiver has **not** sent NAK for `frame_expected`

A Sliding Window Protocol Using Selective Repeat
Adjusting timers

- Sender timeout at sender should be *slightly larger* than roundtrip time
- Problem if round trip time is variable
- If reverse traffic is sporadic, ACK will be irregular, making it hard for receiver to estimate roundtrip time

- Receiver auxiliary ACK timer should be appreciably **shorter** than sender retransmit timeout

Why Restart ACK timer For Out of Order packet?

- Consider the following sequence of events
  - Sender sends packets 0,1,2,3
  - Receiver receives packets 0,1,2,3
  - Receiver sends ACK for all of them and advances window to 4,5,6,7
  - All ACKs are lost
  - *How will the protocol recover?*

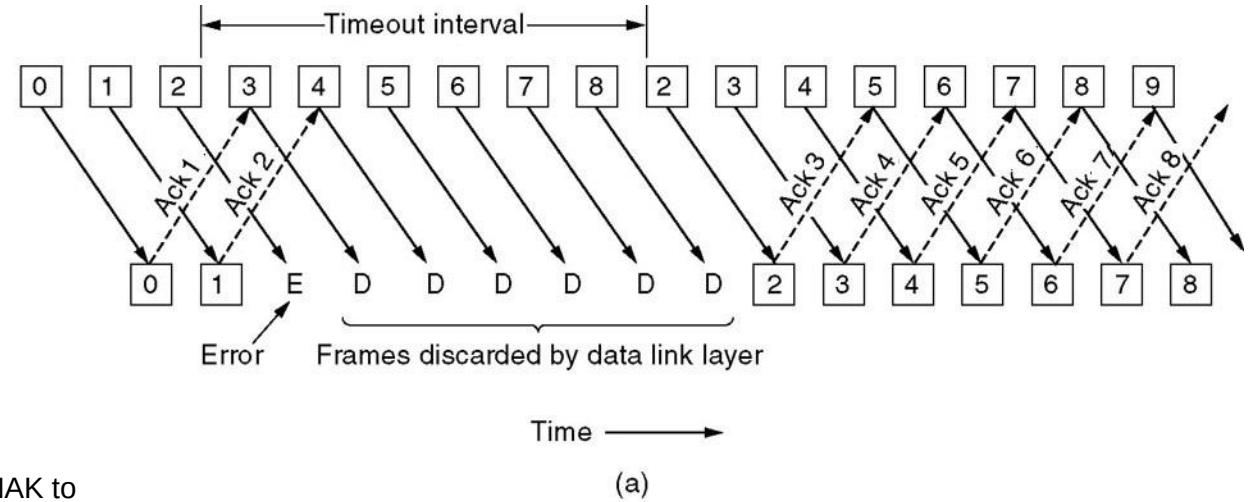
Go Back N vs. Selective Repeat

- 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
    ⇒ **Cumulative** ACK
  - 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 timeout
  - Receiver needs to *buffer multiple frames* up to window size
- *Tradeoff between buffer space and link bandwidth utilization*

Go Back N vs. Selective Repeat

Expected frame not last received frame

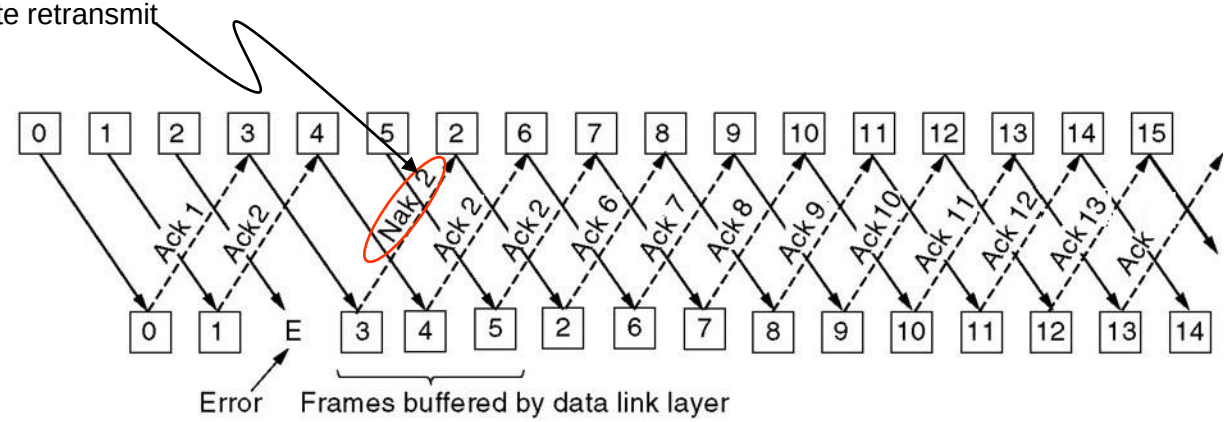
Go Back N



(a)

Use NAK to stimulate retransmit

Selective Repeat with NAK



(b)

Example Data Link Protocols

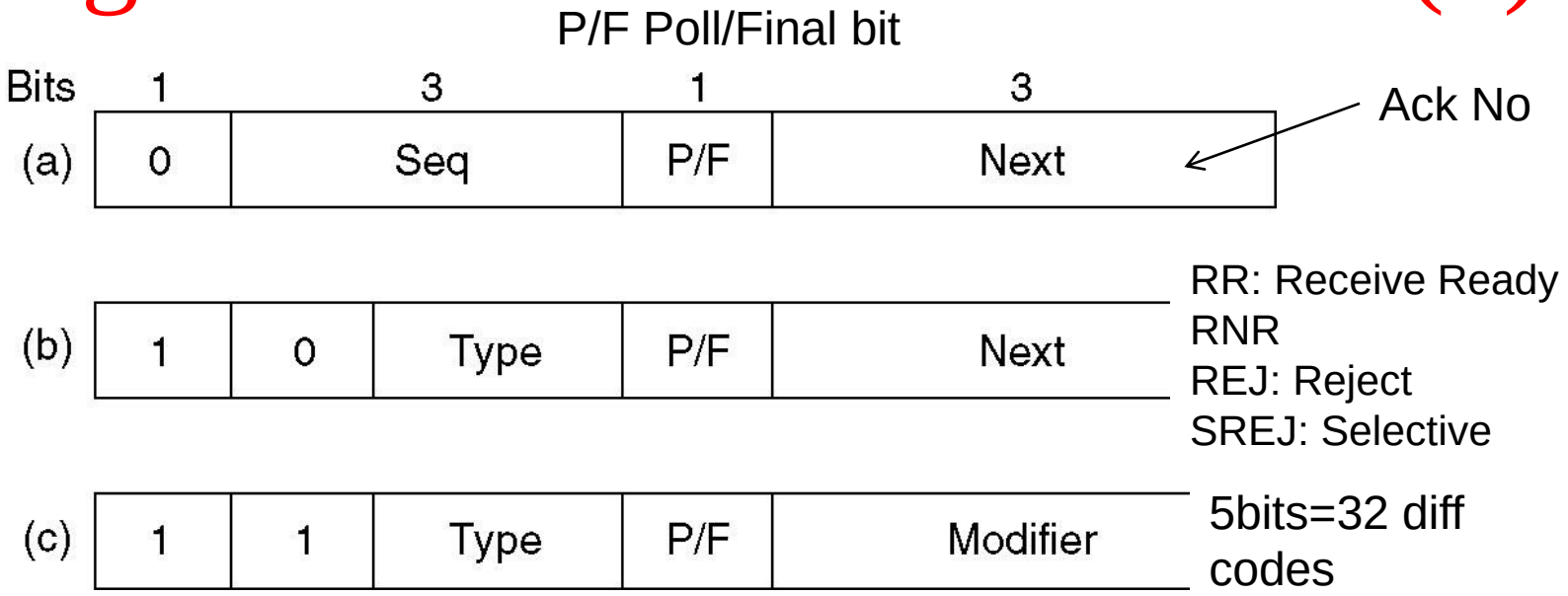
- HDLC – High-Level Data Link Control
- PPP - The Data Link Layer in the Internet

High-Level Data Link Control

- Frame format for bit-oriented protocols.
- HDLC uses bit stuffing and allows non-integer frames of, say, 30.25 bytes
- HDLC provides reliable transmission with a sliding window

Bits	8	8	8	≥ 0	16	8
	0 1 1 1 1 1 1 0	Address	Control	Data	Checksum	0 1 1 1 1 1 1 0

High-Level Data Link Control (2)



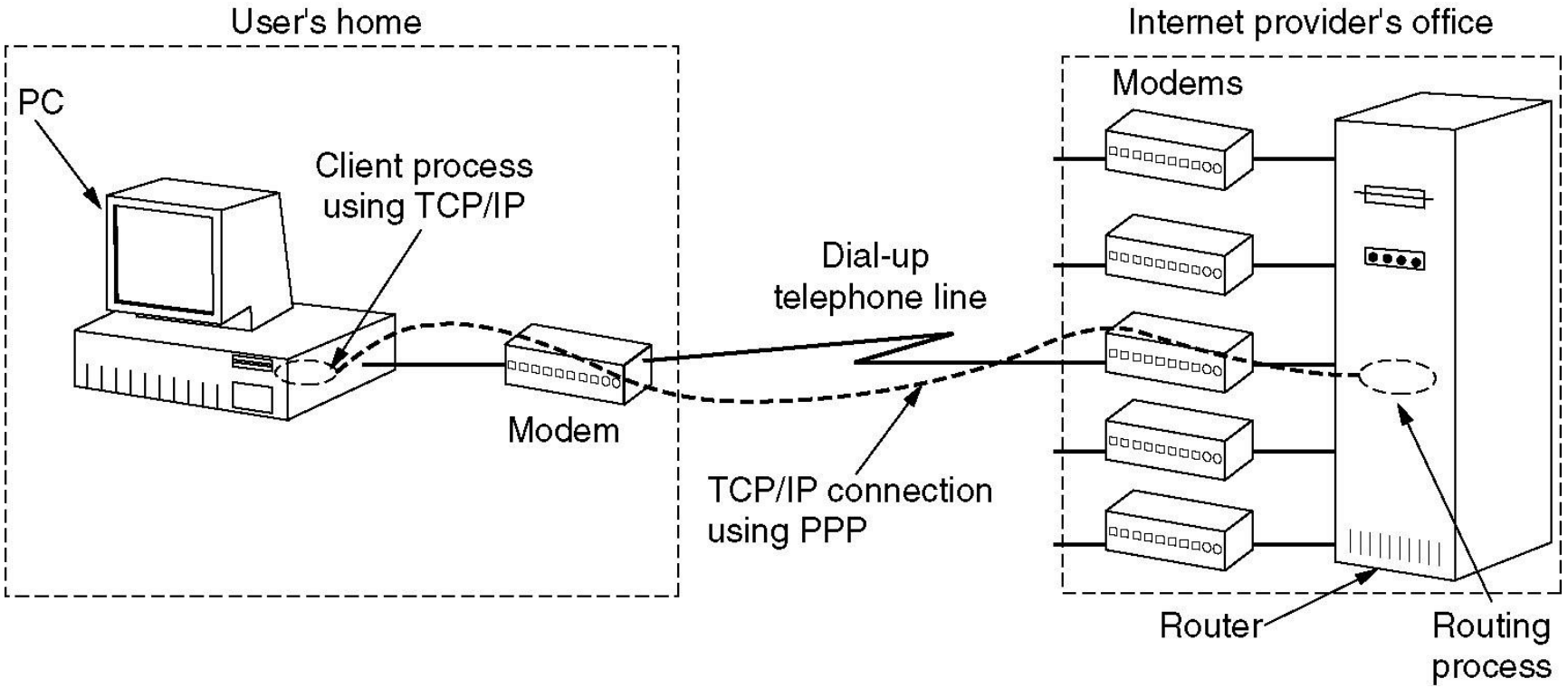
(a) An information frame starts with 0.

(b) A supervisory frame 10: for flow and error control when piggybacking is not required. Control functions such as acknowledgment of frames, requests for re-transmission, and requests for temporary suspension of frames being transmitted

(c) An unnumbered frame 11: also used for control purposes. It is used to perform link initialization, link disconnection and other link control functions.

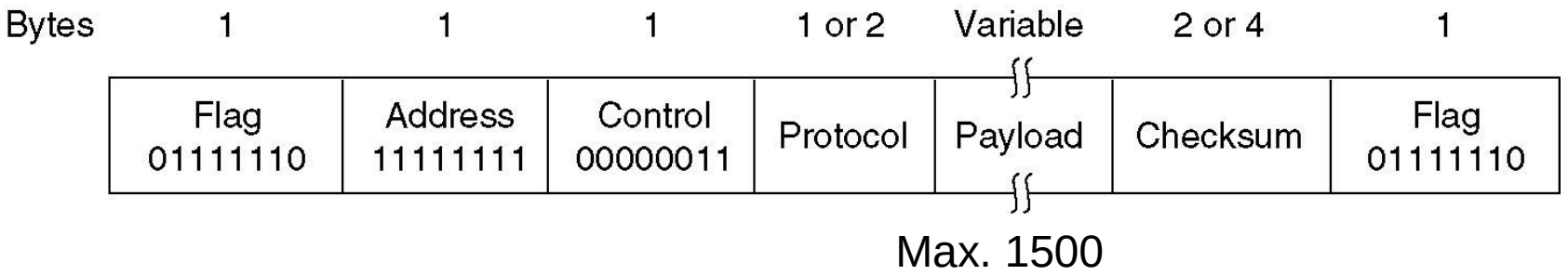
The Data Link Layer in the Internet

A home personal computer acting as an internet host.



PPP – Point to Point Protocol

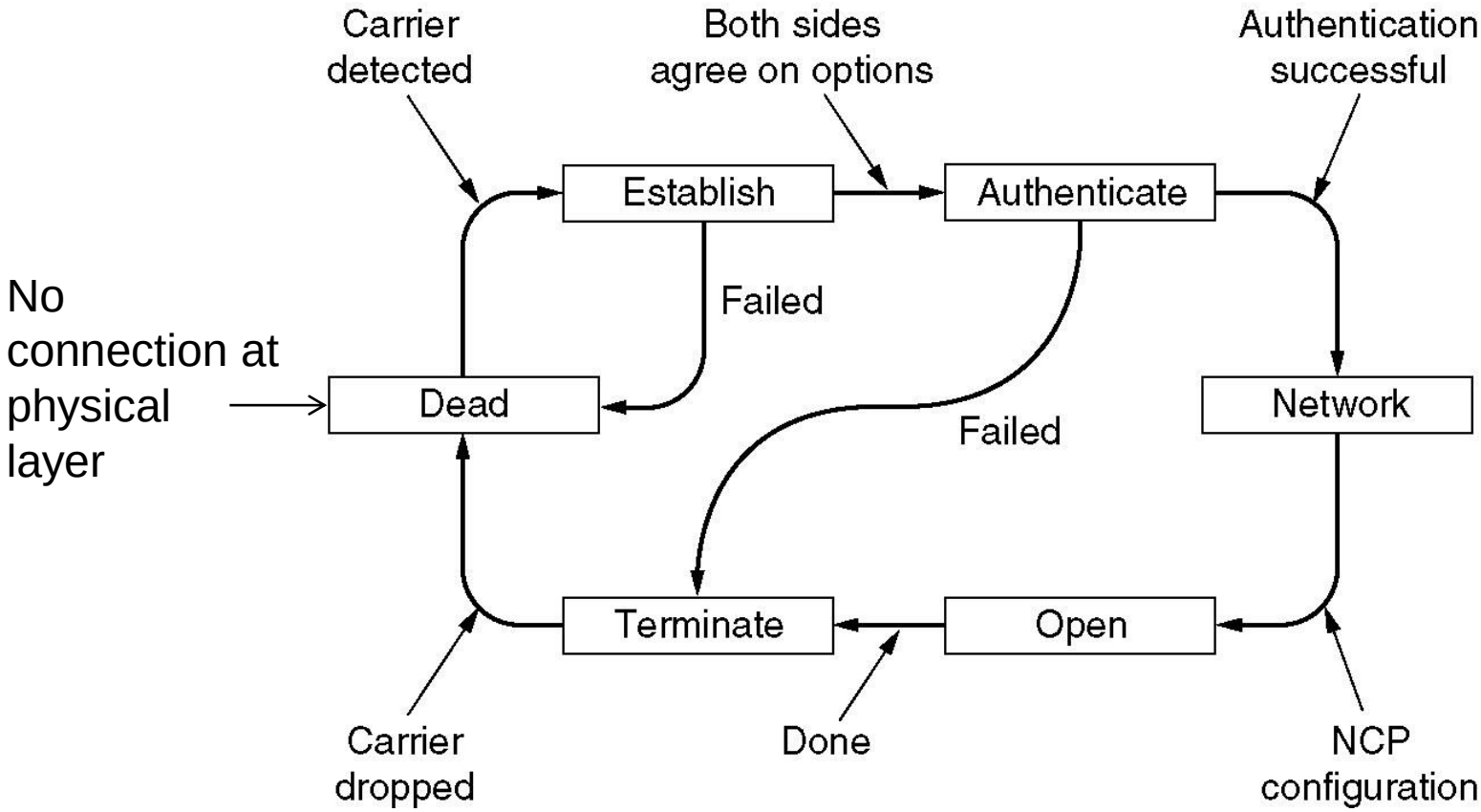
- The PPP full frame format for unnumbered mode operation.
- PPP uses byte stuffing and all frames are an integral number of bytes



Address field: This field is always set to the binary value 11111111 to indicate that all stations are to accept the frame
Control field: the default value of which is 00000011. This value indicates an unnumbered frame
Protocol field: To tell what kind of packet is in the Payload field

- Codes starting with 0 for information
- Codes starting with 1 are used for configurations (Link Control Protocol LCP, Network control Protocol NCP)

PPP – Point to Point Protocol (2)



A simplified phase diagram for bring a line up and down.

PPP – Point to Point Protocol (3)

The LCP frame types.

Name	Direction	Description
Configure-request	I → R	List of proposed options and values
Configure-ack	I ← R	All options are accepted
Configure-nak	I ← R	Some options are not accepted
Configure-reject	I ← R	Some options are not negotiable
Terminate-request	I → R	Request to shut the line down
Terminate-ack	I ← R	OK, line shut down
Code-reject	I ← R	Unknown request received
Protocol-reject	I ← R	Unknown protocol requested
Echo-request	I → R	Please send this frame back
Echo-reply	I ← R	Here is the frame back
Discard-request	I → R	Just discard this frame (for testing)