Data Link
Protocols 2
sliding window protocols

Piggybacking

- When send data → send with it ACK for the next expected frame
  - Advantage: save bandwidth, packet processing capacity
  - Problem: If no data to send→ no ACK will be sent so will keep resending same data waiting for its ACK.
  - Solution: Use small timeout at the receiver of the data if it has no data to send , it times out and send an empty packet as ACK
    - Condition: timeout to send ACK <time out to retransmit
    - Why?? Good design since did not allow even one retransmission since goal of retransmission is: if packet is lost or corrupted

Sliding Window Protocols

- A One-Bit Sliding Window Protocol
- A Protocol Using Go Back N
- A Protocol Using Selective Repeat

- Protocols are ***full duplex***
- ACK is ***piggy-backed*** with data

Sliding Window Protocols

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

- As sender:
  - Take packet from Network layer give it next available sequence number in sending window.
  - If received ACK on lower end of window advance it.
  - If sent all from 15→ 19 cannot send another packet until receive ACK on lower end(15)

- As receiver:

  - If received packet 3013 → do not send ACK
  - If received 3011→ send ACK 3012 and advance lower end

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 */
```

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

Continued →

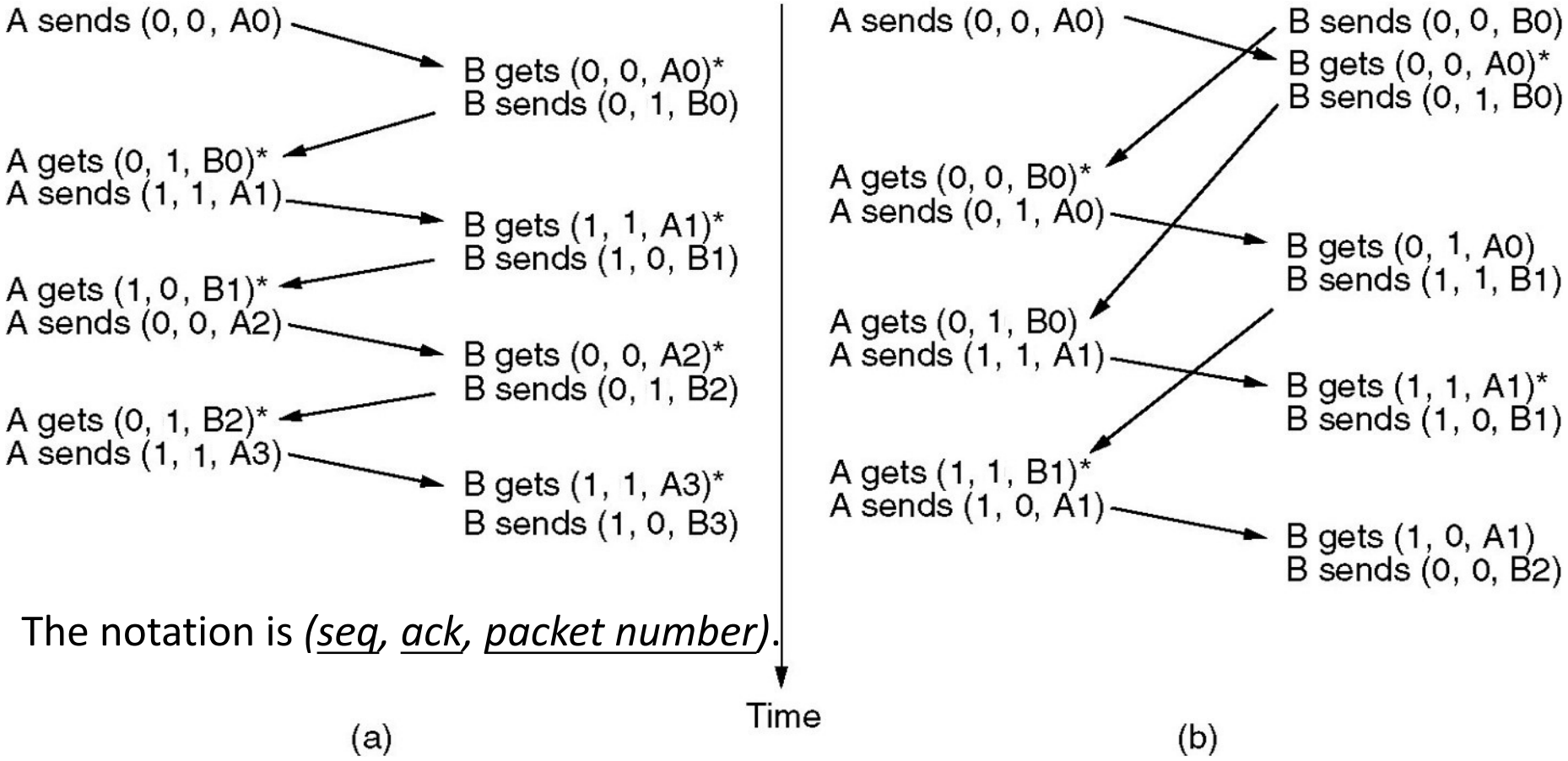
```
while (true) {
    wait_for_event(&event);                 /* frame_arrival, cksum_err, or timeout */
    if (event == frame_arrival) {           /* a frame has arrived undamaged. */
        from_physical_layer(&r);            /* go get it */

        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 */
  }
}
```

*ACK is the **circular**(frame expected-1)*

- 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?*



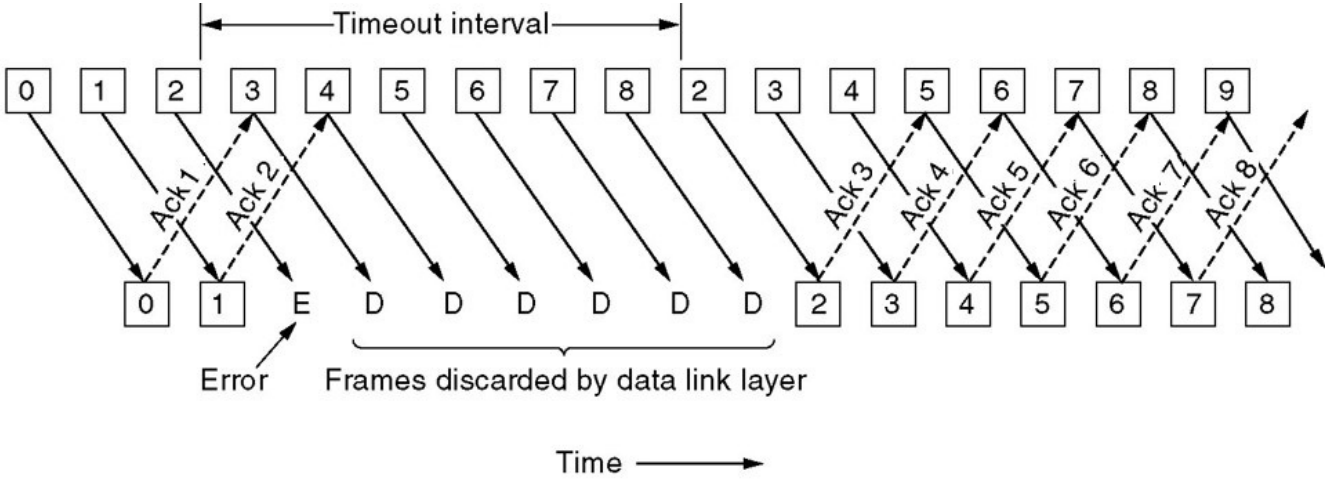
Two scenarios for protocol 4:

(a) Normal case: If one side sends a packet and is received before the other side sends a packet, the protocol works perfectly

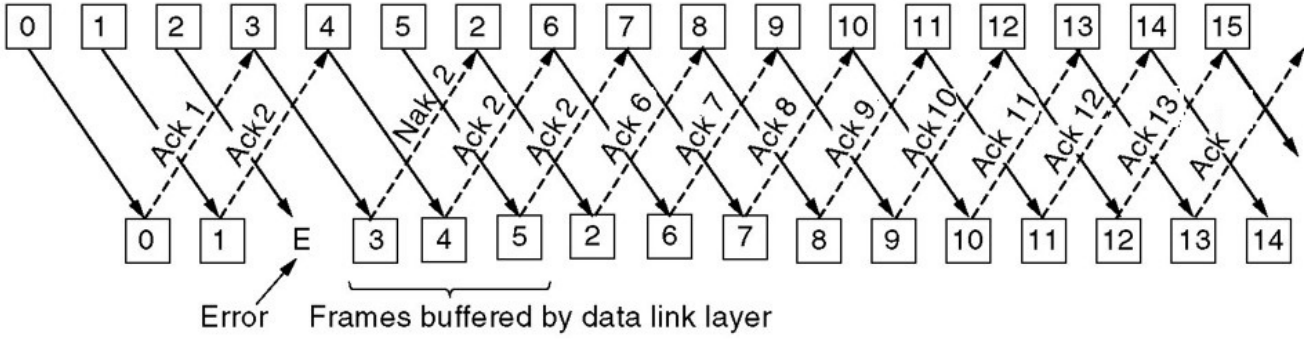
(b) Abnormal case: If both sides send at the same time and the frames cross

- Every packet is sent twice
- Protocol is still correct, but bandwidth is wasted

Go Back N vs. Selective Repeat



(a)



(b)

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);
}

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   );
 to_physical_layer(&s);                 /* transmit the frame */
 start_timer(frame_nr);                 /* start the timer running */
}
```

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

- **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 **cannot** rely on **duplicate** 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 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*

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

⇒ increase sender's window

- Advance ***Lower end*** of ***receiver*** window
- *Remember* receiver window has **fixed size 1**

⇒ 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();
}
}
```

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

- 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

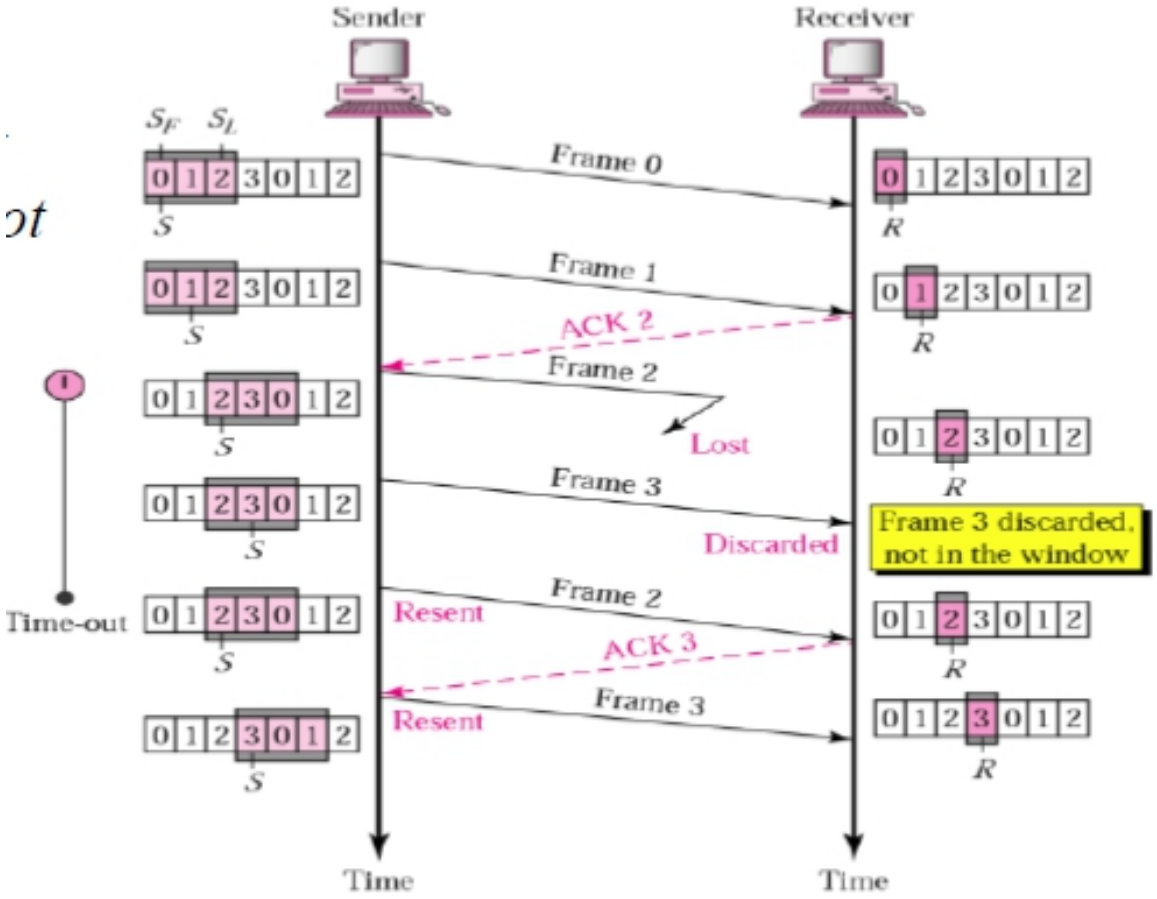
- Assume timeout because of frame loss
- Retransmit all frames ***starting*** from the ***last ACKed*** frame because receiver has window size one

- 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
- I.e. there are at most MAX_SEQ+1 distinct sequence numbers

Sliding Window Protocol Using Go Back N

- If there is no reverse traffic, protocol fails..why?
- Sender window size (maximum outstanding frames) ***must be*** MAX_SEQ and ***not*** MAX_SEQ+1…why?

Sliding Window Protocol Using Go Back N



A Sliding Window Protocol Using Selective Repeat

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 );
  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***

Initially, ***receiver*** window is **fixed** between **0** and **NR_BUFs**

Continued →

A Sliding Window Protocol Using Selective Repeat (3)

```
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

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**

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

```
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

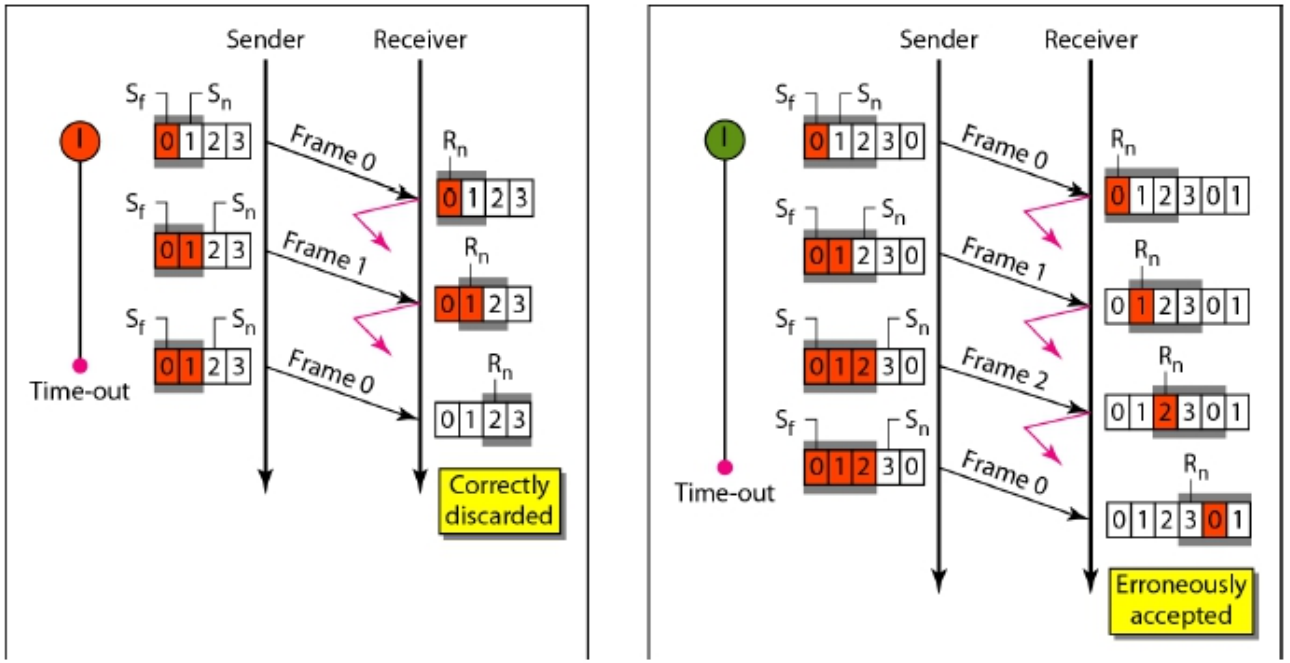
Window size is NR_BUF ⇒ half of MAX_SEQ

A Sliding Window Protocol Using Selective Repeat (5)

- Why is NR_BUFS defined (MAX_SEQ+1)/2?
- Why ACK timer<<retransmit timer?

A Sliding Window Protocol Using Selective Repeat (5)

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



Link utilization

- For selective repeat WS= (2^m /2) =((MAXSEQ+1)/2)
- For Go Back N WS at sender =(2^m -1)=(MAXSEQ) , at receiver WS=1.
- *Efficiency for sliding window protocols =100%*
- WS = *(Tt+2Tp)/Tt*
- *Minimum number of bits to distinguish between frames if not stated explicitly which sliding protocol is used =ceil(log2(WS))*