Data Link Control

Functional requirements for effective data communications between two directly connected transmitting-receiving stations:

- Frame Synchronization
- Use of a Variety of Line Configurations
- Flow Control
- Error Control
- Addressing
- Control and Data on Same Link
- Link Management

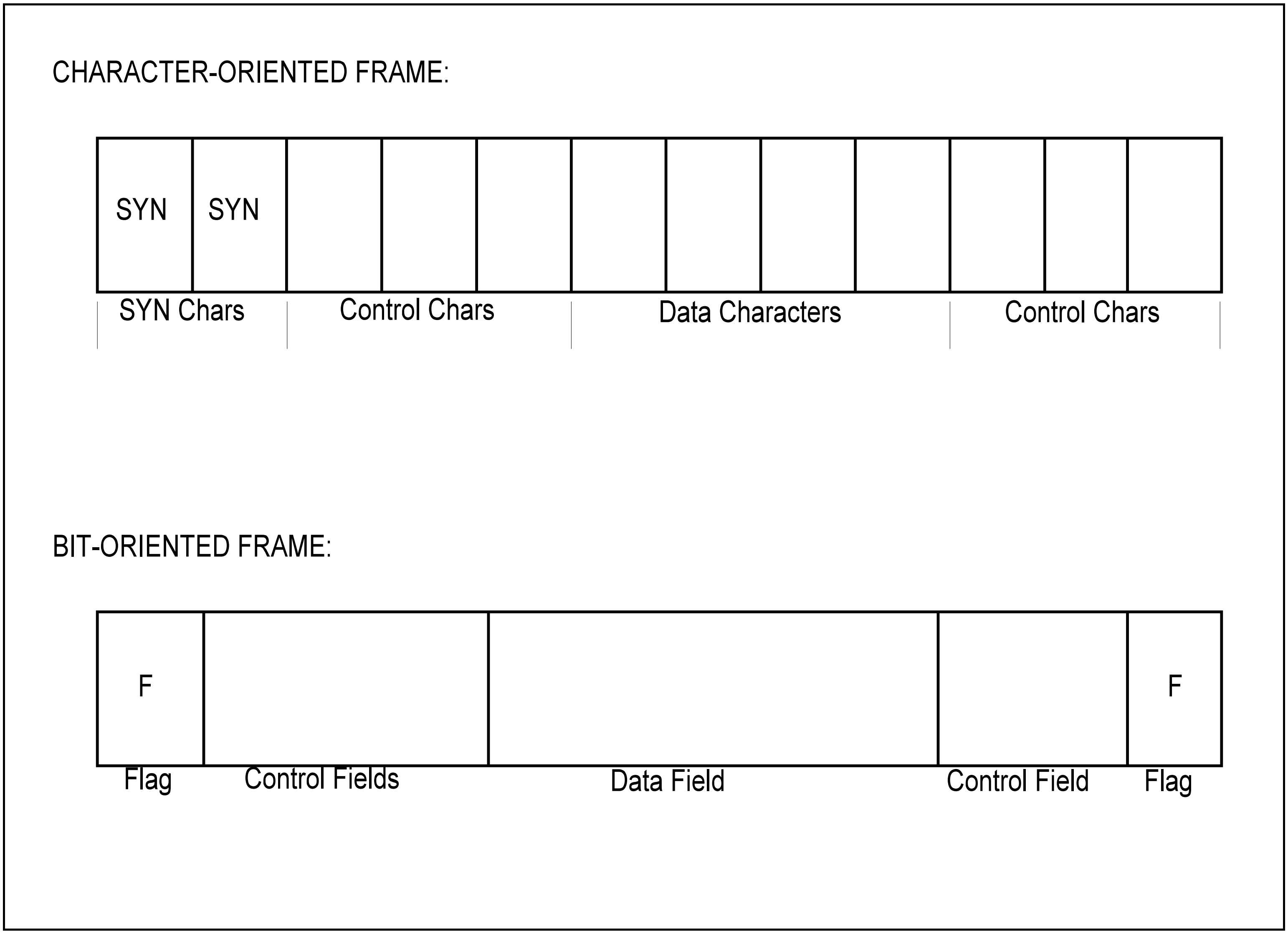
Frame Synchronization

- Character Oriented Frames

  Base unit of information is a character.

- Bit Oriented Frames

  Base unit of information is a bit.



Data Types and Functions for Subsequent Protocols

```
#define MAX_PKT 1024                       /* determines packet size in bytes */

typedef enum {false, true} boolean;        /* boolean type */
typedef unsigned int seq_nr;               /* sequence or ack numbers */
typedef struct {unsigned char data[MAX_PKT];} packet;/*    packet definition */
typedef enum {data, ack, nak} frame_kind;  /* frame_kind definition */

typedef struct {                           /* frames are transported in this layer */
  frame_kind kind;                         /* what kind of a frame is it? */
  seq_nr seq;                              /* sequence number */
  seq_nr ack;                              /* acknowledgement number */
  packet info;                             /* the network layer packet */
} frame;

/* Wait for an event to happen; return its type in event. */
void wait_for_event(event_type  *event);

/* Fetch a packet from the network layer for transmission on the channel. */
void from_network_layer(packet *p);

/* Deliver information from an inbound frame to the network layer. */
void to_network_layer(packet *p);

/* Go get an inbound frame from the physical layer and copy it to r. */
void from_physical_layer(frame *r);

/* Pass the frame to the physical layer for transmission. */
void to_physical_layer(frame *s);

/* Start the clock running and enable the timeout event. */
void start_timer(seq_nr k);

/* Stop the clock and disable the timeout event. */
void stop_timer(seq_nr k);

/* Start an auxiliary timer and enable the ack_timeout event. */
void start_ack_timer(void);

/* Stop the auxiliary timer and disable the ack_timeout event. */
void stop_ack_timer(void);

/* Allow the network layer to cause a network_layer_ready event. */
void enable_network_layer(void);

/* Forbid the network layer from causing a network_layer_ready event. */
void disable_network_layer(void);

/* Macro inc is expanded in-line: Increment k circularly. */
#define inc(k) if (k < MAX_SEQ) k = k + 1; else k = 0
```

An Unrestricted Simplex Protocol

```
/* Protocol 1 (utopia) provides for data transmission in one direction only, from
   sender to receiver.  The communication channel is assumed to be error free
   and the receiver is assumed to be able to process all the input infinitely quickly.
   Consequently, the sender just sits in a loop pumping data out onto the line as
   fast as it can. */

typedef enum {frame_arrival} event_type;
#include "protocol.h"

void sender1(void)
{
  frame s;                          /* buffer for an outbound frame */
  packet buffer;                    /* buffer for an outbound packet */

  while (true) {
      from_network_layer(&buffer);  /* go get something to send */
      s.info = buffer;              /* copy it into s for transmission */
      to_physical_layer(&s);        /* send it on its way */
  }                                 /* Tomorrow, and tomorrow, and tomorrow,
                                       Creeps in this petty pace from day to day
                                       To the last syllable of recorded time.
                                            - Macbeth, V, v */
}


void receiver1(void)
{
  frame r;
  event_type event;                 /* filled in by wait, but not used here */

  while (true) {
      wait_for_event(&event);       /* only possibility is frame_arrival */
      from_physical_layer(&r);      /* go get the inbound frame */
      to_network_layer(&r.info);    /* pass the data to the network layer */
  }
}
```

Flow Control & Error Control Mechanisms

Flow Control:

- Stop-and-Wait
- Sliding Window

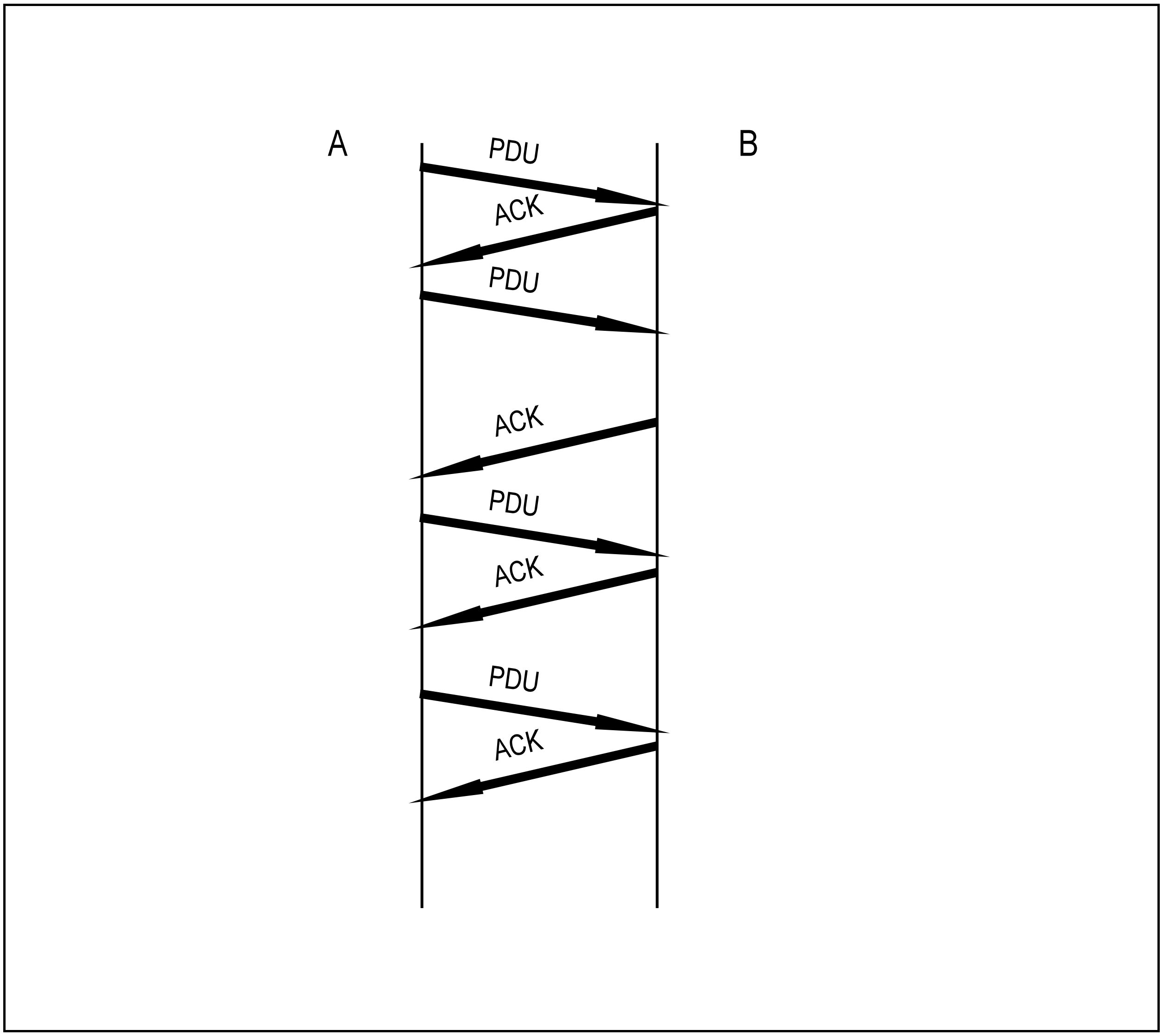
Error Control:

- Stop-and-Wait ARQ
- Go-Back-N ARQ
- Selective-Repeat ARQ

Stop-and-Wait Flow Control Protocol

Basic Operation:

- Sender transmits a data frame and then, before transmitting any other data frames, waits for a signal (an ACK) from the receiver, which indicates its willingness to receive more data.



Stop-and-Wait Flow Control Protocol

A Simplex Stop-and-Wait Protocol

```
/* Protocol 2 (stop-and-wait) also provides for a one-directional flow of data from
   sender to receiver. The communication channel is once again assumed to be error
   free, as in protocol 1. However, this time, the receiver has only a finite buffer
   capacity and a finite processing speed, so the protocol must explicitly prevent
   the sender from flooding the receiver with data faster than it can be handled. */

typedef enum {frame_arrival} event_type;
#include "protocol.h"

void sender2(void)
{
  frame s;                        /* buffer for an outbound frame */
  packet buffer;                  /* buffer for an outbound packet */
  event_type event;               /* frame_arrival is the only possibility */

  while (true) {
      from_network_layer(&buffer); /* go get something to send */
      s.info = buffer;            /* copy it into s for transmission */
      to_physical_layer(&s);      /* bye-bye little frame */
      wait_for_event(&event);     /* do not proceed until given the go ahead */
  }
}

void receiver2(void)
{
  frame r, s;                     /* buffers for frames */
  event_type event;               /* frame_arrival is the only possibility */
  while (true) {
      wait_for_event(&event);     /* only possibility is frame_arrival */
      from_physical_layer(&r);    /* go get the inbound frame */
      to_network_layer(&r.info);  /* pass the data to the network layer */
      to_physical_layer(&s);      /* send a dummy frame to awaken sender */
  }
}
```

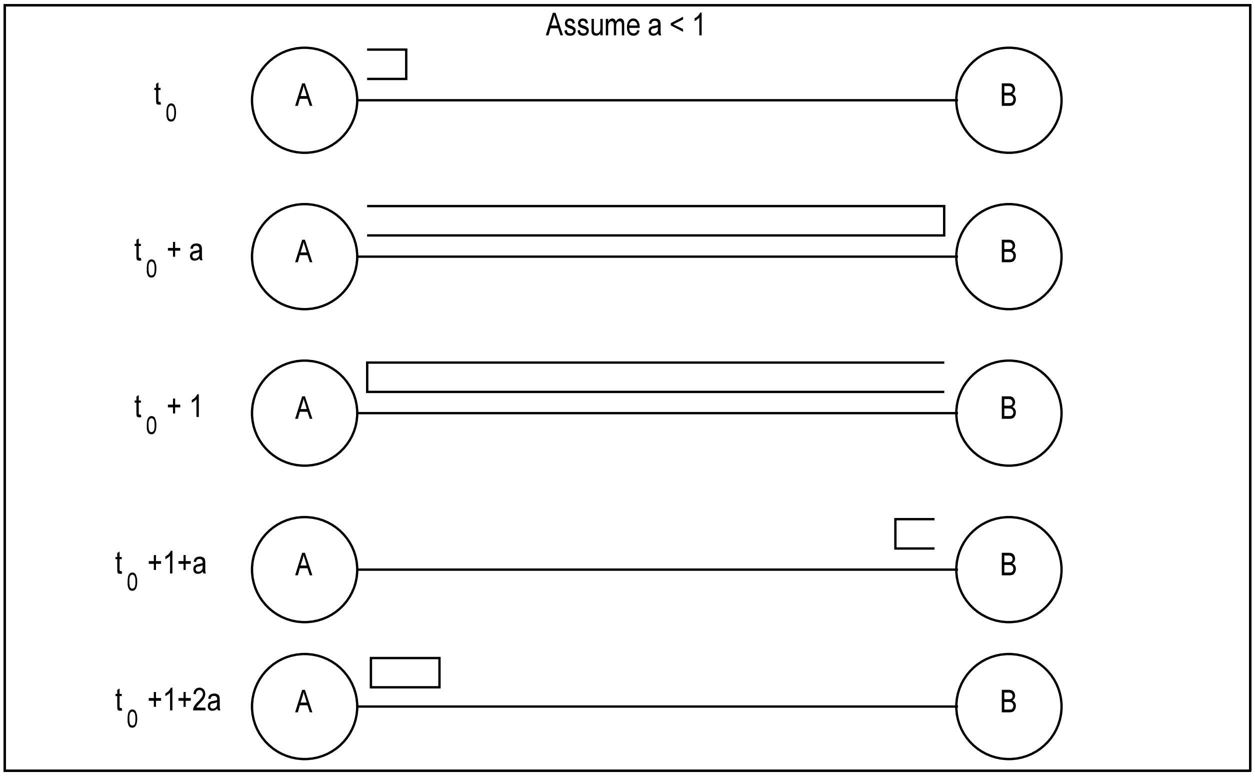
Stop-and-Wait Flow Control - Performance

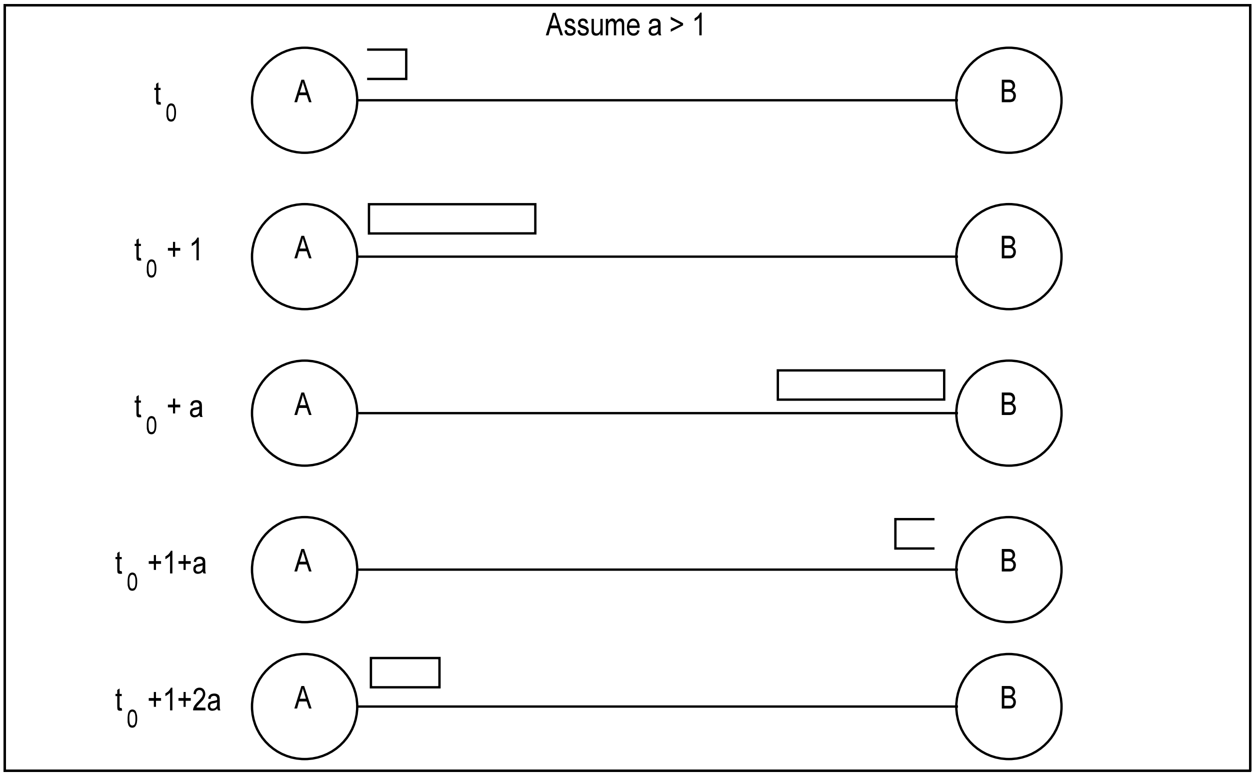
- Assume a half-duplex, point-to-point link
- It can be shown that the efficiency of the link is

$$U = 1/(1 + 2a)$$

where

a = Bit Propagation Time/Frame Transmission Time


Assume a < 1

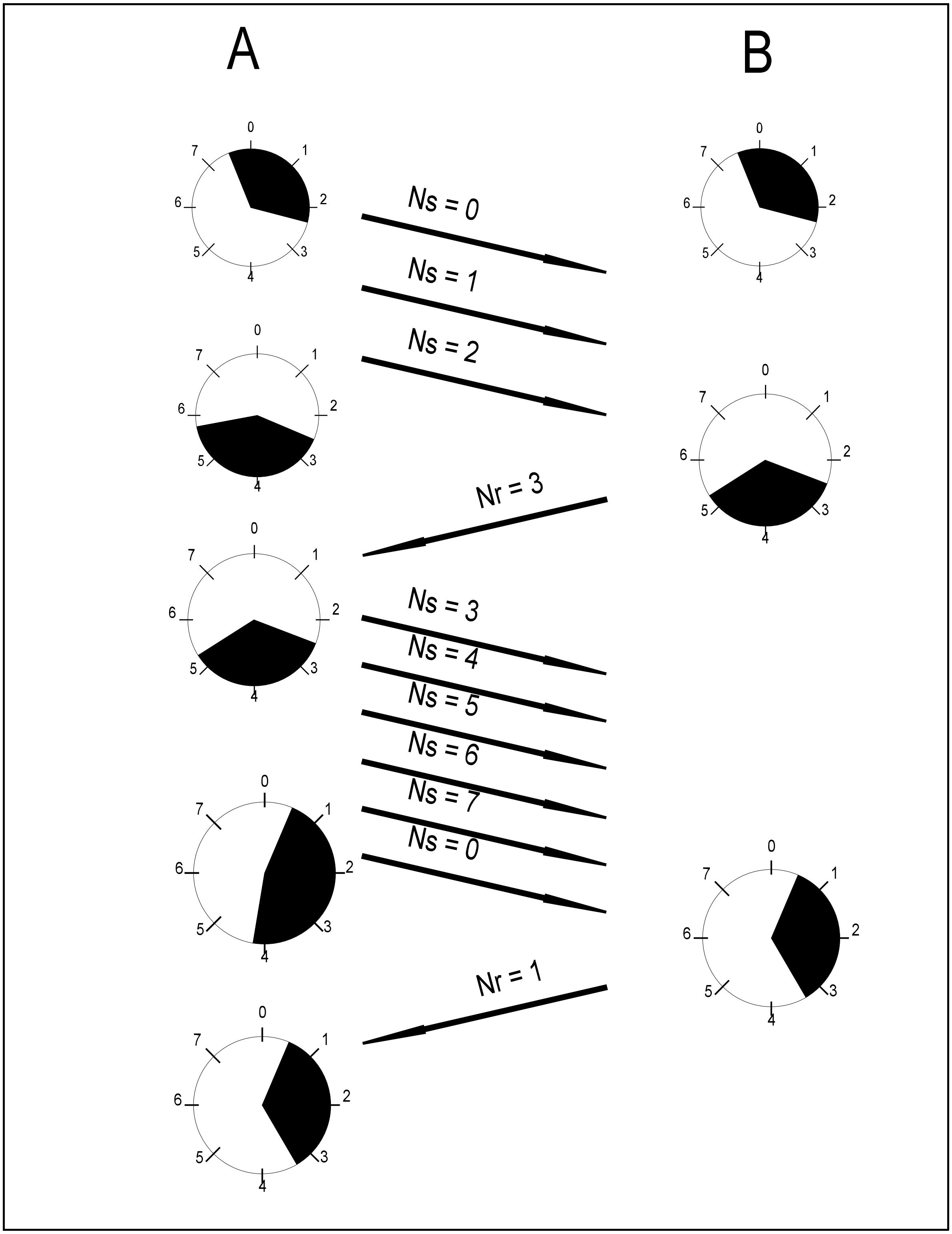

Assume a > 1

Sliding Window Protocol

Basic Operation:

- The sender and receiver each maintain a window.
  - Transmit Window:
    A list of the sequence numbers of frames that a transmitting station may send.
  - Receive Window:
    A list of the sequence numbers of frames that a receiving station is willing to receive.
- The windows open and close (slide) as frames are transmitted and received.
- The Maximum Transmit Window Size (N) determines how many frames can be in transit simultaneously.
- The Window Size can be used as a flow control mechanism.

Sliding Window Protocol



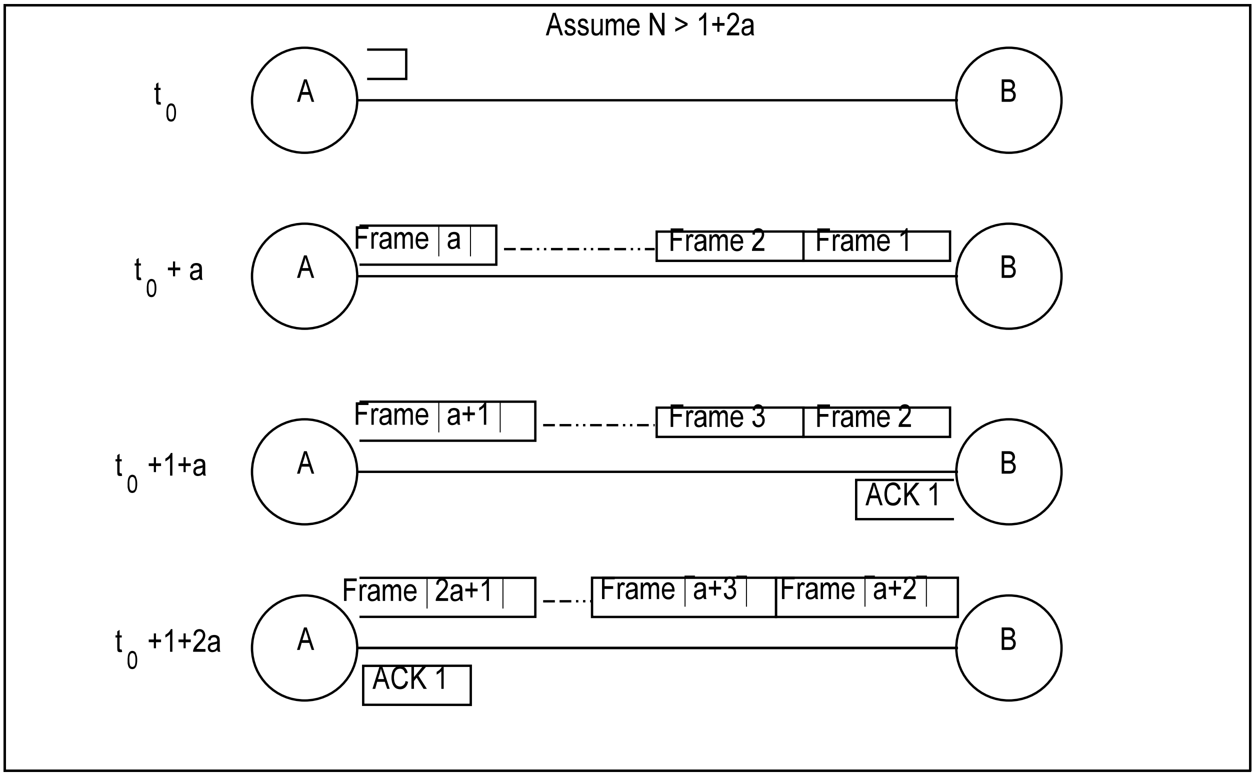
Sliding Window Protocol

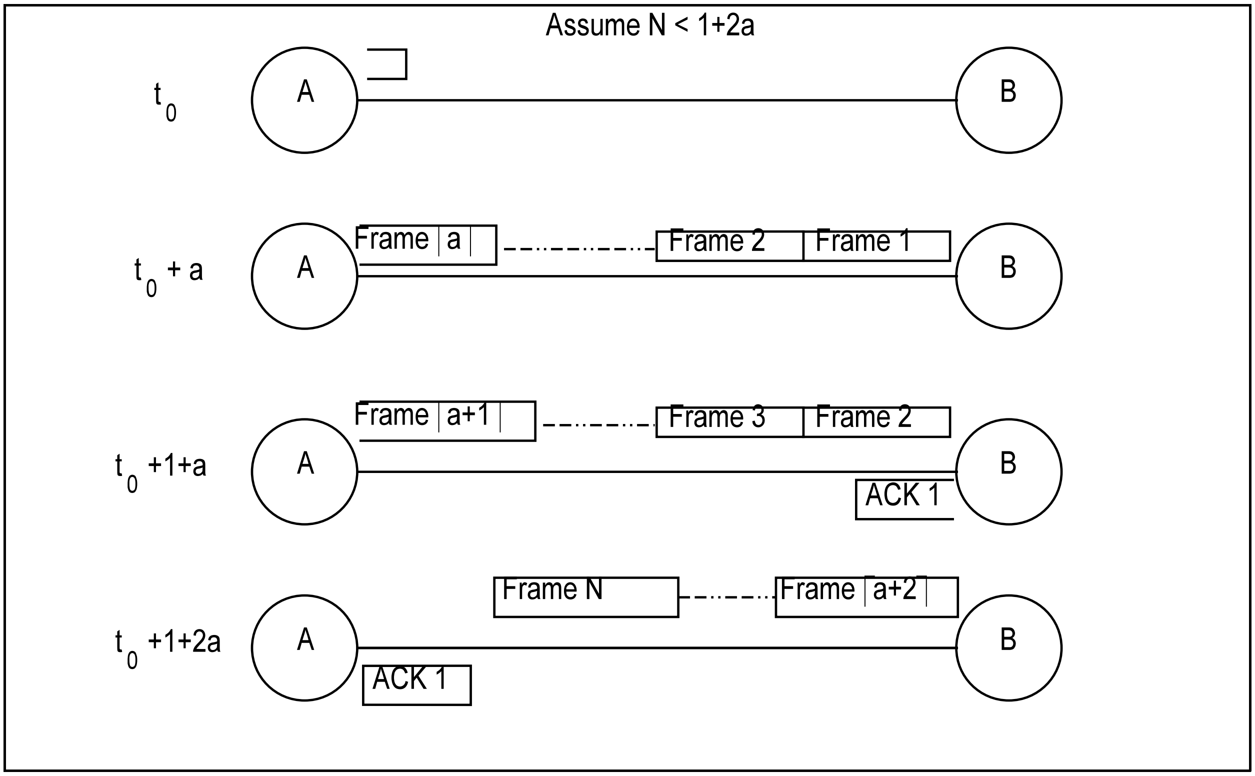
Characteristics & Performance

- The Maximum Transmit Window Size (N) determines how many frames can be in transit simultaneously.
- The Window Size can be used as a flow control mechanism.
- Efficiency is now a function of both 'N' and 'a'.

$$U = 1 \quad \text{for } N > 2a+1$$

$$U = N/(2a+1) \quad \text{for } N < 2a+1$$


Assume N > 1+2a
t 0
A
B
t 0 + a
Frame a
Frame 2
Frame 1
t 0 +1+a
Frame a+1
Frame 3
Frame 2
ACK 1
t 0 +1+2a
Frame 2a+1
Frame a+3
Frame a+2
ACK 1


Assume N < 1+2a
t 0
A
B
t 0 + a
Frame a
Frame 2
Frame 1
t 0 +1+a
Frame a+1
Frame 3
Frame 2
ACK 1
t 0 +1+2a
Frame N
Frame a+2
ACK 1

Error Control

Types of Errors:

- Lost Frame:
  The receiver is not aware that a frame has been transmitted.
- Damaged Frame:
  A recognizable frame arrives at the receiver, however, some of the bits are in error.

Error Control Ingredients:

- Error Detection:
  Ascertain that there was an error.
- Positive Acknowledgement:
  The receiver ACKs successfully received, error-free frames.
- Retransmission After Timeout:
  The source retransmits a frame that has not been acknowledged after a predetermined amount of time.
- Negative Acknowledgement and Retransmission:
  The destination NAKs frames in which an error is detected. The source retransmits such frames.

Automatic Repeat Request (ARQ) Techniques:

- Stop-and-Wait ARQ
- Go-Back-N Continuous ARQ
- Selective-Repeat Continuous ARQ

Stop-and-Wait ARQ

Uses a Simple Stop-and-Wait Acknowledgement Scheme:

Receiver sends either an ACK or a NAK (see code).

Questions & Problems:

1. What if the transmitted frame is lost?

   Answer: Must equip the transmitter with a TIMER.

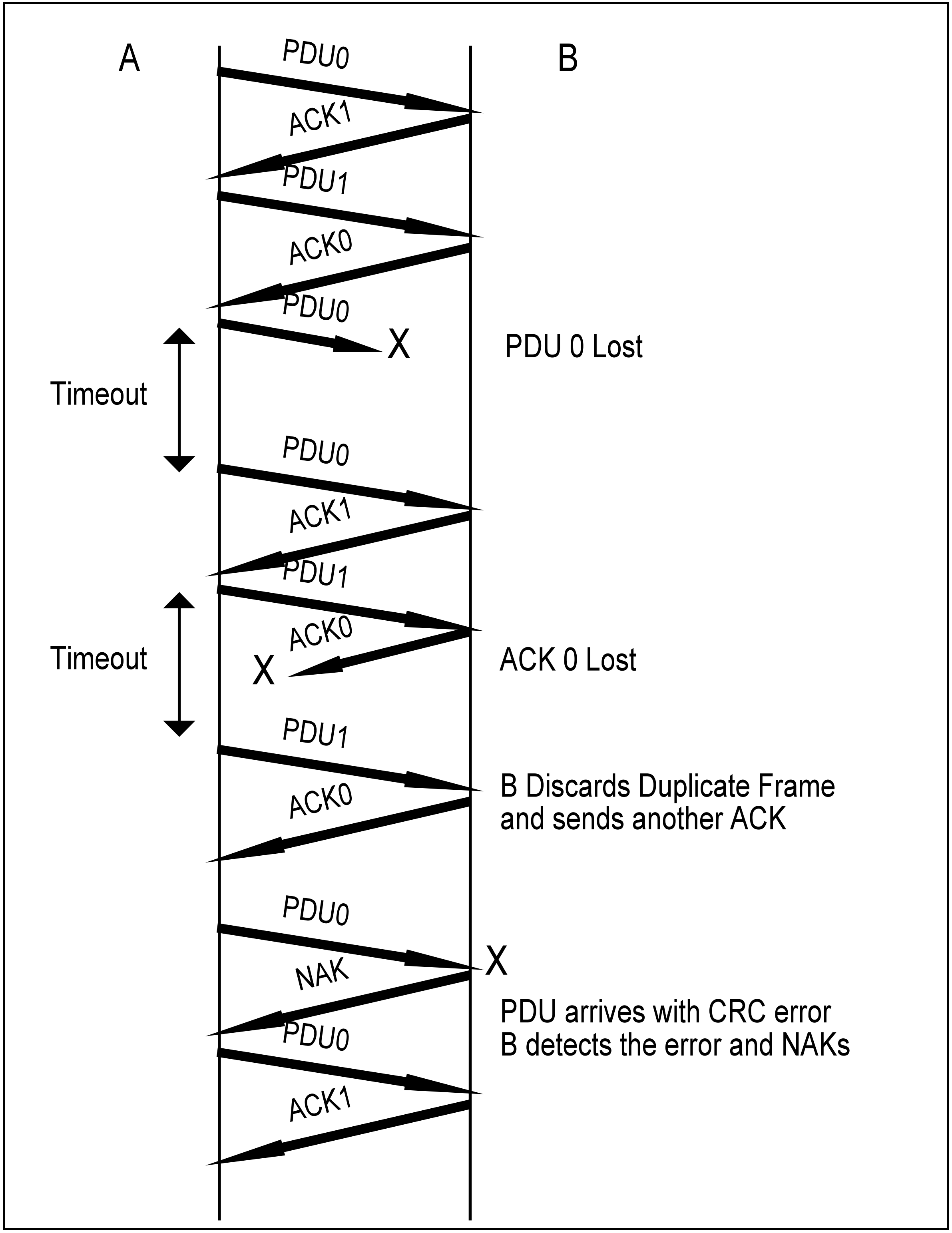
2. What if the ACK gets lost (receiver may get a duplicate)?

   Answer: Must put SEQUENCE NUMBERS (names) within the transmitted frames to allow the receiver to detect duplicates.

3. a) The transmitter sends out frame 0 which is properly received.
   b) The ACK is slow in getting back to the transmitter. The timer goes off causing the transmitter to send a duplicate of frame 0 which is also received correctly.
   c) The receiver sends back a second ACK.
   d) The first ACK arrives at the transmitter which then sends out frame 1 which gets lost.
   e) Here comes the second ACK.

   Answer: Must sequence the ACKs (e.g. ACK0 - ACK1).

Stop-and-Wait ARQ



Stop-and-Wait ARQ

A Simplex Stop-and-Wait ARQ Protocol

```
/* Protocol 3 (par) allows unidirectional data flow over an unreliable channel. */

#define MAX_SEQ 1                       /* must be 1 for protocol 3 */
typedef enum  {frame_arrival, cksum_err, timeout} event_type;
#include "protocol.h"

void sender3(void)
{
  seq_nr next_frame_to_send;            /* seq number of next outgoing frame */
  frame s;                              /* scratch variable */
  packet buffer;                        /* buffer for an outbound packet */
  event_type event;

  next_frame_to_send = 0;               /* initialize outbound sequence numbers */
  from_network_layer(&buffer);          /* fetch first packet */
  while (true) {
        s.info = buffer;                /* construct a frame for transmission */
        s.seq = next_frame_to_send;     /* insert sequence number in frame */
        to_physical_layer(&s);          /* send it on its way */
        start_timer(s.seq);             /* if answer takes too long, time out */
        wait_for_event(&event);         /* frame_arrival, cksum_err, timeout */
        if (event == frame_arrival) {
                from_physical_layer(&s);  /* get the acknowledgement */
                if (s.ack == next_frame_to_send) {
                        stop_timer(s.ack);      /* turn the timer off */
                        from_network_layer(&buffer);    /* get the next one to send */
                        inc(next_frame_to_send);    /* invert next_frame_to_send */
                }
        }
  }
}

void receiver3(void)
{
  seq_nr frame_expected;
  frame r, s;
  event_type event;

  frame_expected = 0;
  while (true) {
        wait_for_event(&event);         /* possibilities: frame_arrival, cksum_err */
        if (event == frame_arrival) {   /* a valid frame has arrived. */
                from_physical_layer(&r);  /* go get the newly arrived frame */
                if (r.seq == frame_expected) {/* this is what we have been waiting for. */
                        to_network_layer(&r.info);    /* pass the data to the network layer */
                        inc(frame_expected); /* next time expect the other sequence nr */
                }
                s.ack = 1 – frame_expected;   /* tell which frame is being acked */
                to_physical_layer(&s);        /* send acknowledgement */
        }
  }
}
```

Go-back-N ARQ

- A station may send a series of frames determined by the window size, using the sliding window flow control technique.

- While no errors occur, the destination station will ACK incoming frames as usual.

Error Control:

Assume that A is sending frames to B.

1. Damaged or Lost Frames:

   a. A transcribes frame i.

   B detects an error in frame i and has previously successfully received frame i-1.

   B sends a NAK i, indicating that frame i is rejected.

   When A receives this NAK, it must go back and retransmit frame i and all subsequent frames that it has transmitted.

   b. Frame i is lost in transit.

   A subsequently sends frame i+ 1.

   B receives frame i+ 1 out of order, and sends a NAK i.

   Upon receipt of the NAK, A proceeds as above.

c. Frame i is lost in transit and A does not soon send additional frames.
   B receives nothing and returns neither an ACK nor a NAK.
   A will time out and retransmit frame i.

2. Damaged or Lost ACK:

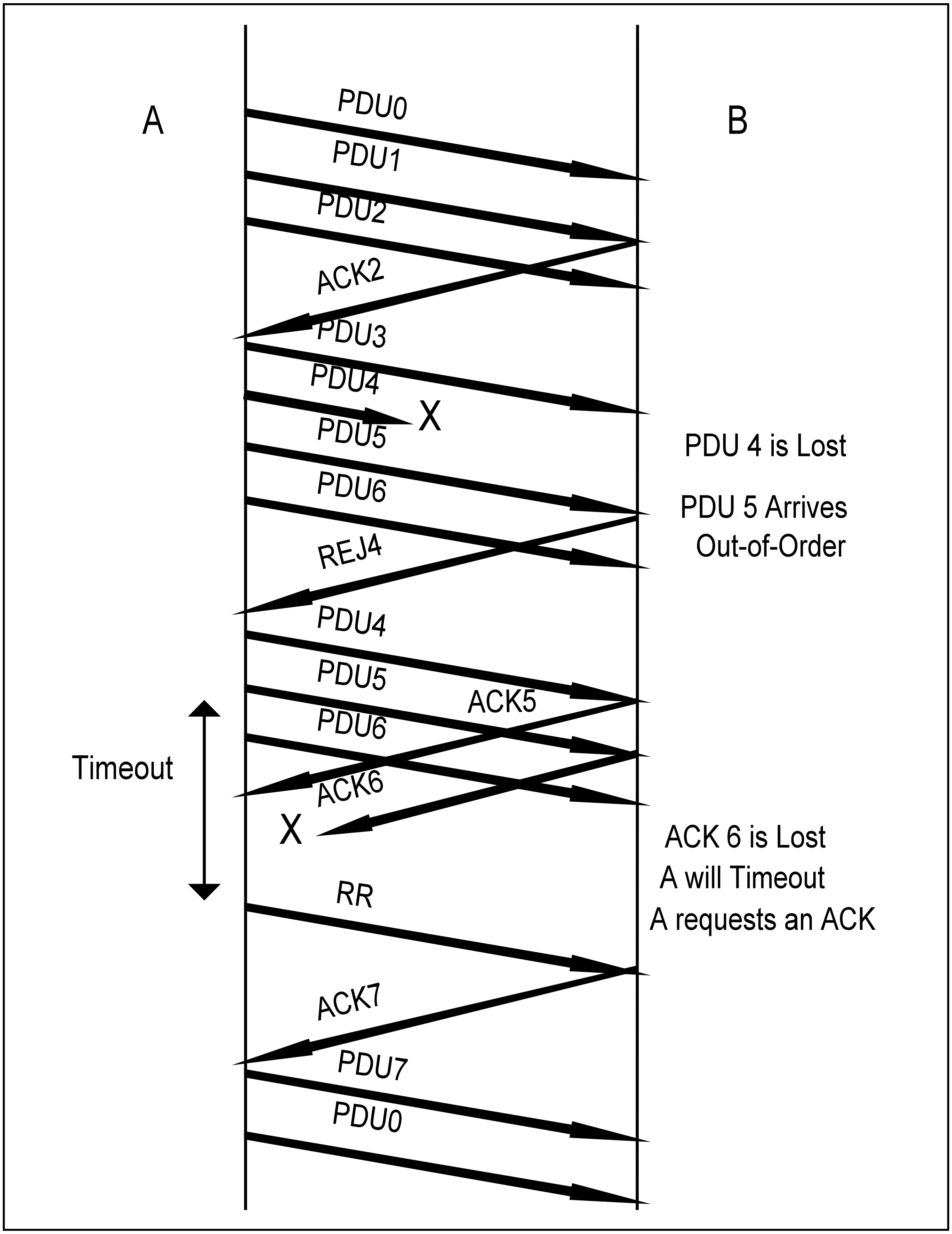
   a. B receives frame i and sends ACK (i+ 1), which is lost in transit.
      Since ACKs are cumulative (e.g., ACK 6 means that all frames through 5 are acknowledged,) it may be that A will receive a subsequent ACK to a subsequent frame that will do the job of the lost ACK before the associated timer expires.

   b. If A's timer expires, A retransmits frame i and all subsequent frames (or alternately, A can create a checkpoint by requesting an ACK).

3. Damaged or Lost NAK:

   If a NAK is lost, A will eventually time out on the associated frame and retransmit that frame and all subsequent frames.

Go-Back-N ARQ



Go-Back-N ARQ

Go-Back-N ARQ

Questions & Problems:

1. What is the maximum size of the receive window?

   Ans. 1

2. What is the maximum size of the transmit window?

   Ans. 2^n-1

Proof:

a) Assume window size is 2^n.
b) Sender transmits all 2^n frames.
c) An ACK comes in for frame 2^n.
d) There is no way to tell if all the frames have arrived correctly or none of the frames have arrived correctly.

A Bidirectional 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 = 1 – frame_expected;           /* piggybacked ack */
  to_physical_layer(&s);                /* transmit the frame */
  start_timer(s.seq);                   /* start the timer running */
  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 == 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 sender's sequence number */
          }
      }
      s.info = buffer;                  /* construct outbound frame */
      s.seq = next_frame_to_send;       /* insert sequence number into it */
      s.ack = 1 – frame_expected;       /* seq number of last received frame */
      to_physical_layer(&s);            /* transmit a frame */
      start_timer(s.seq);               /* start the timer running */
  }
}
```

A Bidirectional Go-Back-N ARQ Protocol

```
/* Protocol 5 (go back n) allows multiple outstanding frames. The sender may transmit up
   to MAX_SEQ frames without waiting for an ack. In addition, unlike in 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 */
}

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 */
```

A Bidirectional Go-Back-N ARQ Protocol cont.

```
 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 */
           }

           /* 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();
 }
}
```

Selective-Repeat ARQ

- A station may send a series of frames determined by the window size, using the sliding window flow control technique.
- While no errors occur, the destination station will ACK incoming frames as usual.

Error Control:

Assume that A is sending frames to B.

1. Damaged or Lost Frames:

   a. A transmits frame i.
   B detects an error in frame i and has previously successfully received frame i-1.
   B sends a NAK i, indicating that frame i is rejected.
   B buffers any subsequent frames received that are within its receiving window.
   When A receives this NAK, it retransmits frame i only.

   b. Frame i is lost in transit.
   A subsequently sends frame i+ 1.
   B receives frame i+ 1 out of order, and sends a NAK i.
   B buffers frame i+ 1 and all subsequent frames within its receiving window.
   When A receives the NAK, it retransmits frame i only.

c. Frame i is lost in transit and A does not soon send additional frames.
B receives nothing and returns neither an ACK nor a NAK.
A will time out and retransmit frame i only.

2. Damaged or Lost ACK:

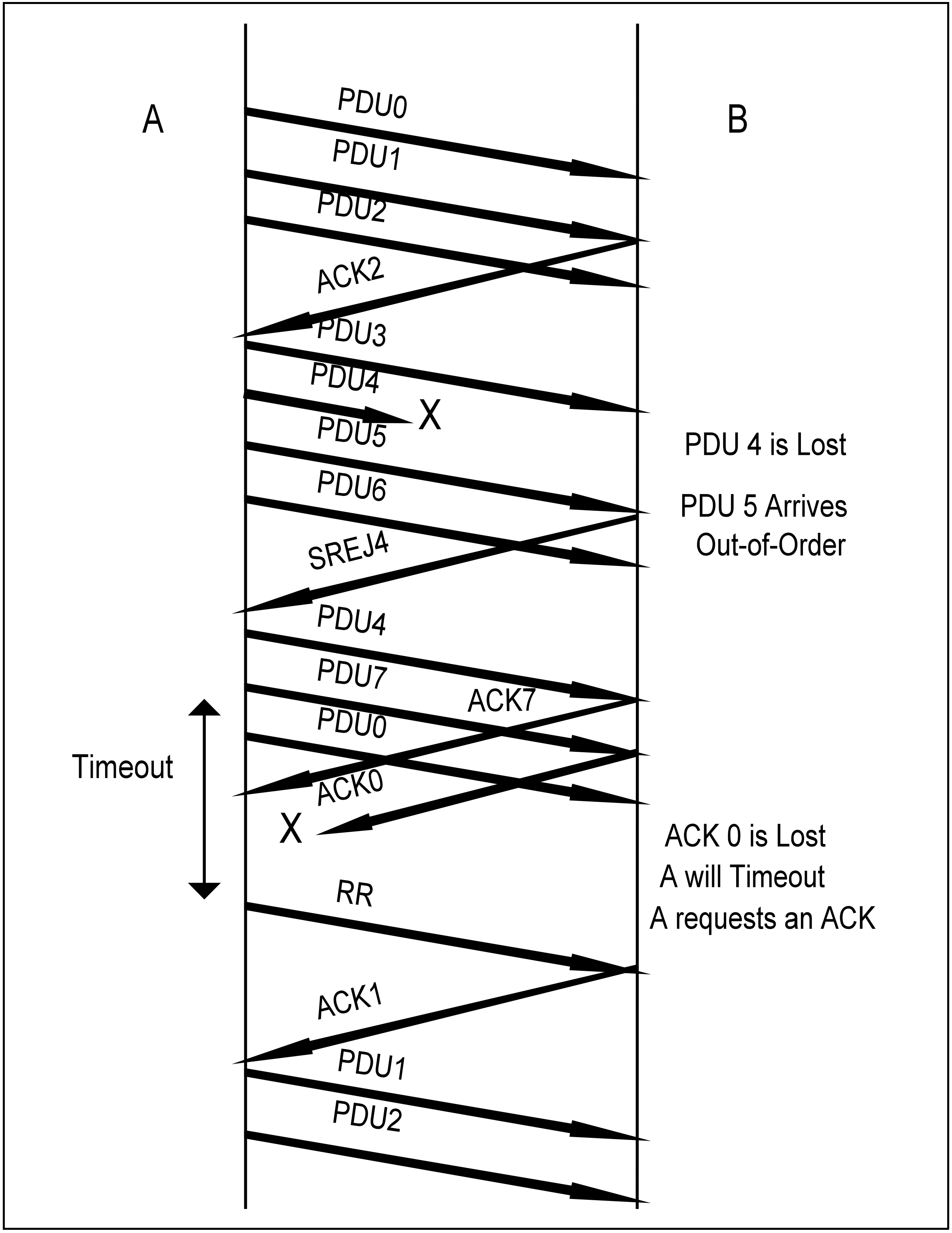
   a. B receives frame i and sends ACK (i+ 1), which is lost in transit.
   Since ACKs are cumulative, it may be that A will receive a subsequent ACK to a subsequent frame that will do the job of the lost ACK before the associated timer expires.

   b. If A's timer expires, A retransmits frame i only (or alternately, A can create a <u>checkpoint</u> by requesting an ACK).

3. Damaged or Lost NAK:

   If a NAK is lost, A will eventually time out on the associated frame and retransmit that frame only.

Selective-Repeat ARQ



Selective-Repeat ARQ

Questions & Problems:

1. What is the maximum size of the receive window?

   Answer: The maximum receive window size should be no more than half the range of sequence numbers.

2. What is the maximum size of the transmit window?

   Answer: Transmit window size + Receive window size $\leq 2^n$

A Bidirectional Selective Repeat ARQ Protocol

```
/* Protocol 6 (selective repeat) 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 2ˆn – 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 */
}

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;
```

A Bidirectional Selective Repeat ARQ Protocol cont.

```
  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 */
                  }
              }
          }
          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();
  }
}
```