計算機ネットワーク

開講クォーター: 1-2Q

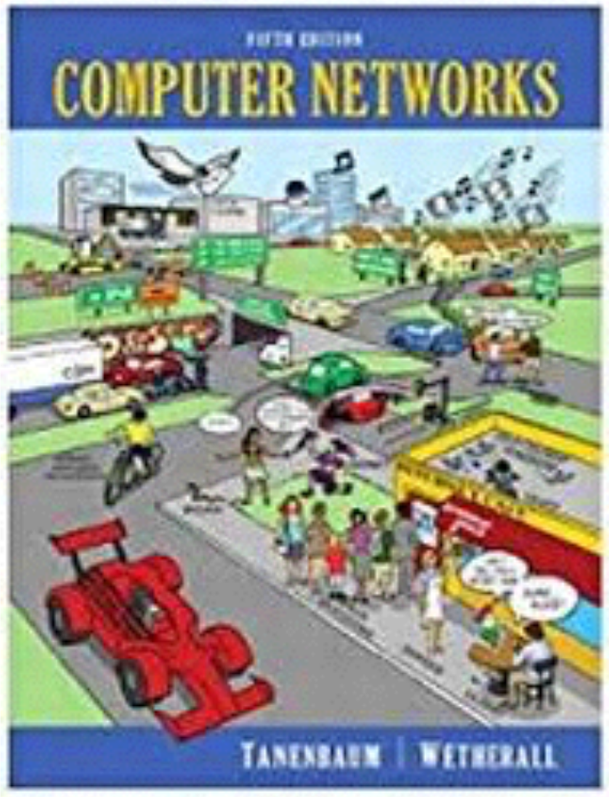
曜日・時限: 火7-8限

講義室: 1Q @ W834, 2Q @ W931

横田理央

rioyokota@gsic.titech.ac.jp

参考書

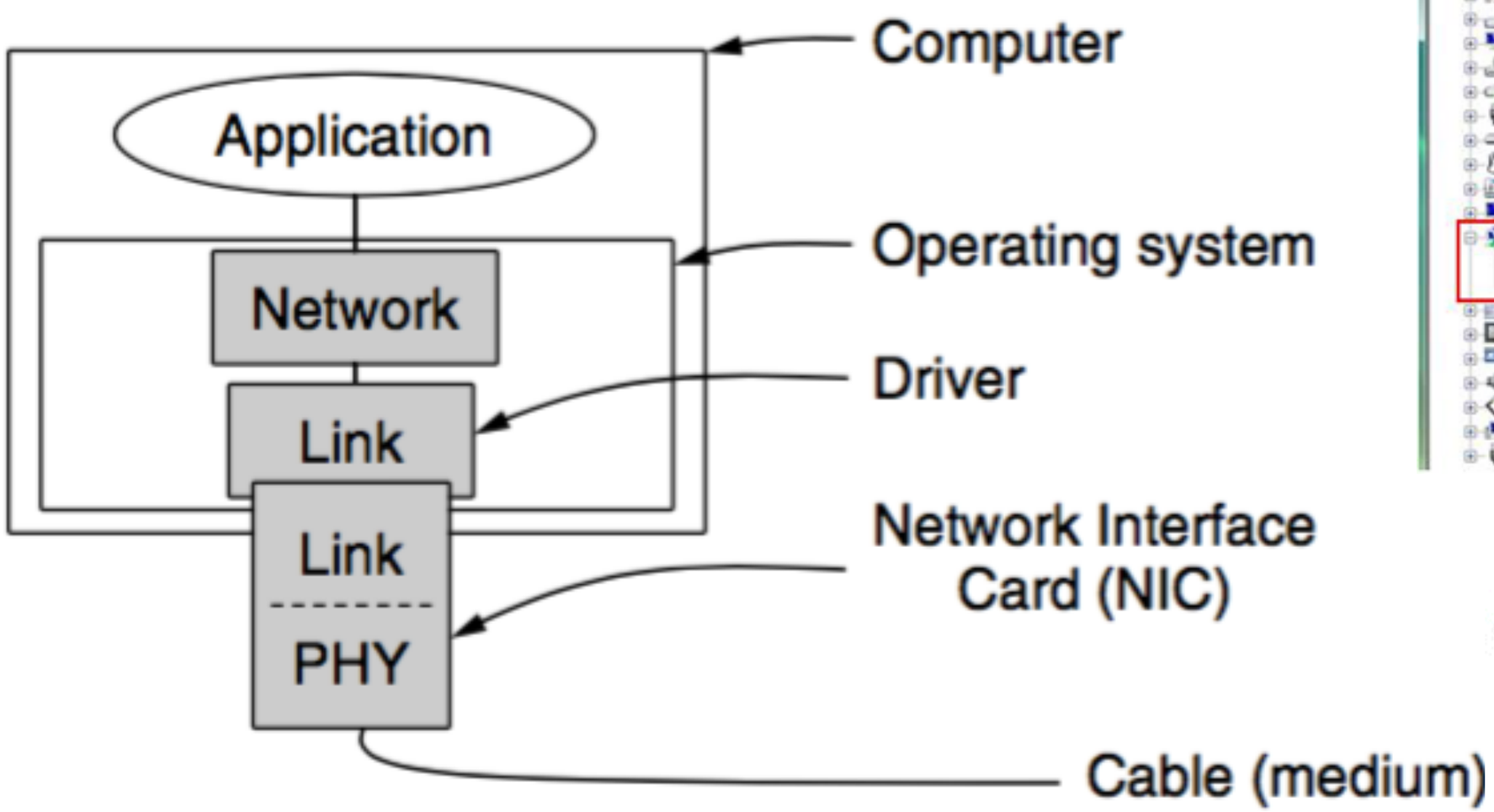


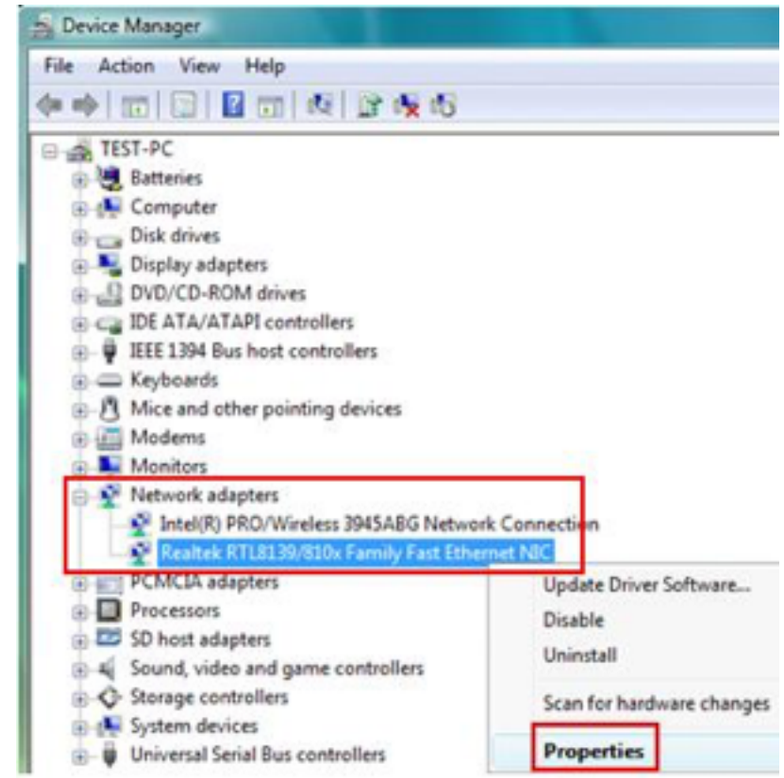
教科書

講義日程（1Q）

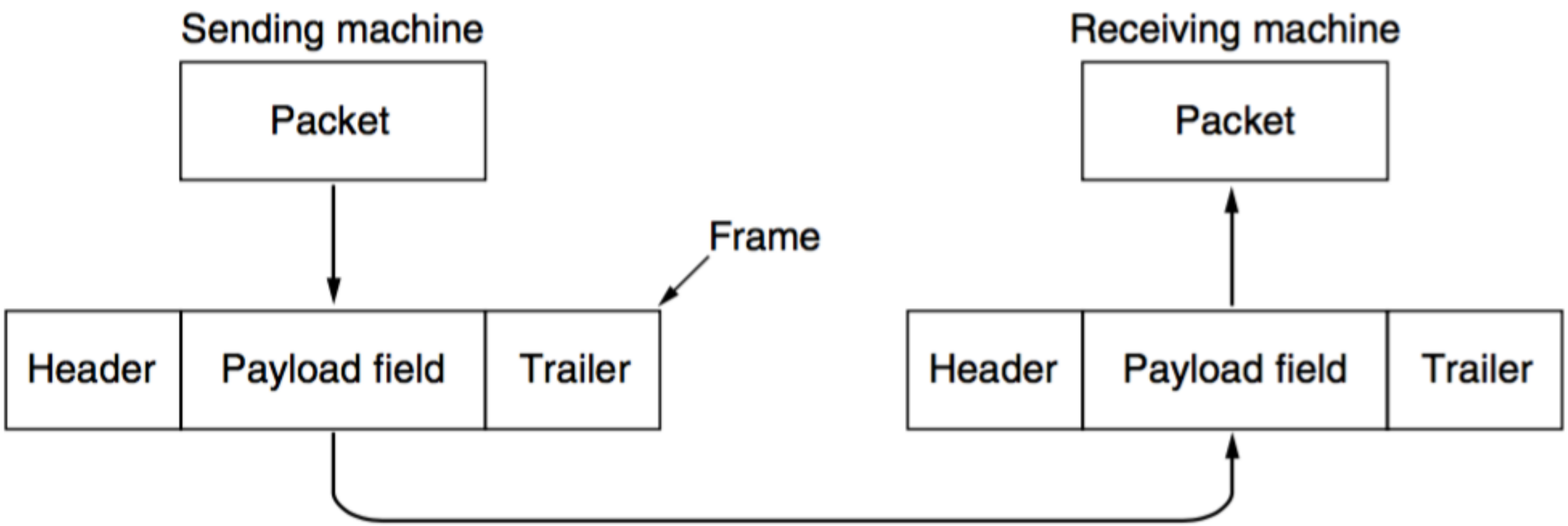
		授業計画		課題
04/05	第１回	計算機ネットワークの基本概念 ハードウェア・ソフトウェア，参照モデル	１章	ネットワークの種類と参照モデルを理解し プロトコル階層と各層の設計課題
04/12	第２回	物理層１ 有線伝送と無線伝送	２章	物理チャネルの特性を理解し データ通信の理論的基礎を理解
04/19	第３回	物理層２ デジタル変調と多重化	２章	ベースバンド伝送と通過帯域伝送， 電話網，携帯電話システムを説明できる
04/26	第４回	データリンク層１ 誤りの検出・訂正	３章	誤りの検出・訂正のしくみを理解し 検出・訂正符号の計算ができる
05/10	第５回	データリンク層２ データリンク・プロトコル	３章	データリンク・プロトコルの種類， 各プロトコルを定量的に評価できる
05/17	第６回	メディア・アクセス副層１ ブロードキャスト・チャネル	４章	多重アクセス・プロトコルを理解し データ・レートを計算できる
05/24	第７回	メディア・アクセス副層２ 無線 LAN，Bluetooth, RFID	４章	個別のプロトコル・スタックを理解し データリンク層スイッチングを理解
05/31	第８回	理解度確認総合演習 (中間試験) 第１回から第７回までの内容の演習形式による確認		第１回から第７回までの理解度確認と 到達度自己評価

Data link layer





Assumptions



1. Sending machine always has data ready to send
2. Machines do not crash
3. Payload field is considered all data
4. There are procedures to compute/check the checksum

protocol.h

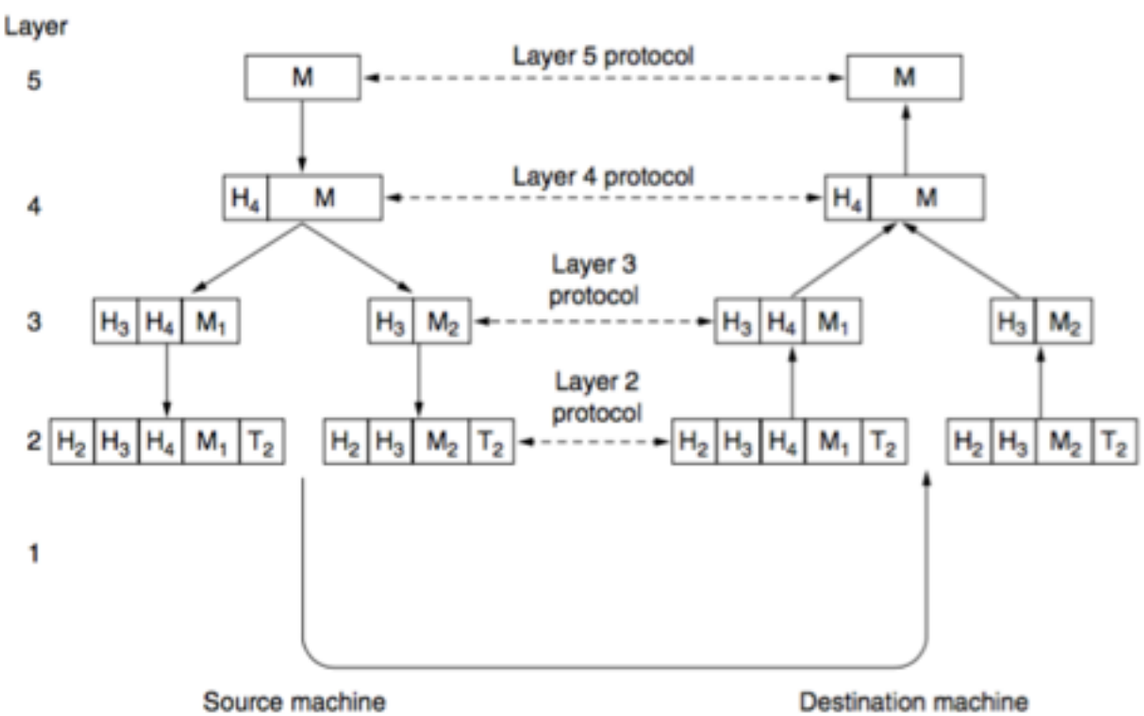
```
#define MAX_PKT 1024                                    /* determines packet size in bytes */

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 frame is it? */
  seq_nr seq;                                           /* sequence number */
  seq_nr ack;                                           /* acknowledgement number */
  packet info;                                          /* the network layer packet */
} frame;
```

seq_nr : small integer used to number the frames
packet : unit of information exchanged between the network layer and the data link layer
frame : first three contain control information and the last contains data
kind : tells whether there are any data in the frame

protocol.h

```
/* 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
```

$event = \{timeout, cksum_err, frame_arrival, network_layer_ready\}$



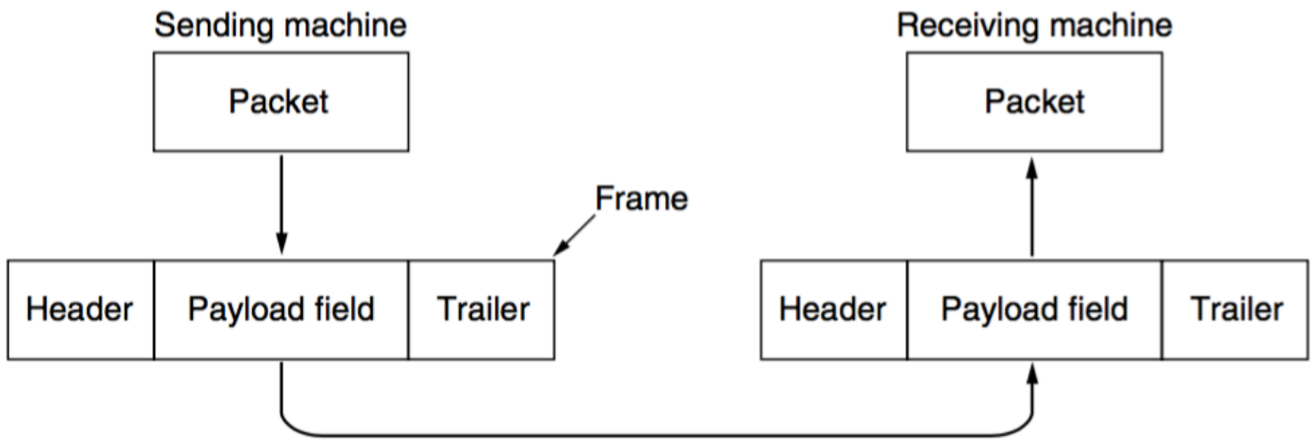
Utopian Simplex Protocol

<u>Assumptions</u>

1. Both the transmitting and receiving network layers are always ready
2. The communication channel between the data link layers never damages or loses frames
3. Data are transmitted in one direction only
4. Processing time can be ignored

No flow control
No error correction

Utopian Simplex Protocol




```
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 */
  }
}

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 */
  }
}
```

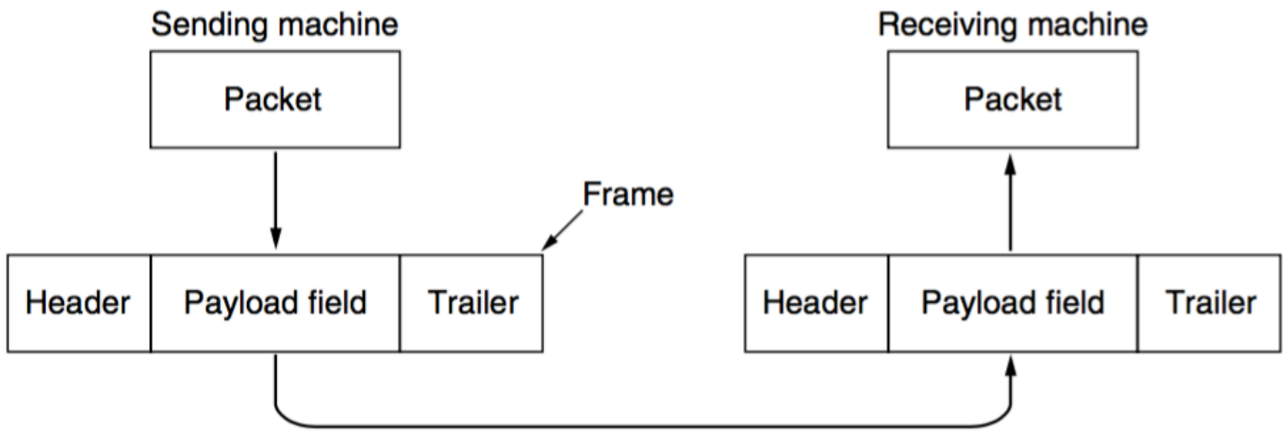
Simplex Stop-and-Wait Protocol

Assumptions

1. ~~Both the transmitting and receiving network layers are always ready~~
2. ~~The communication channel between the data link layers never damages or loses frames~~
3. Data are transmitted in one direction only
4. Processing time can be ignored

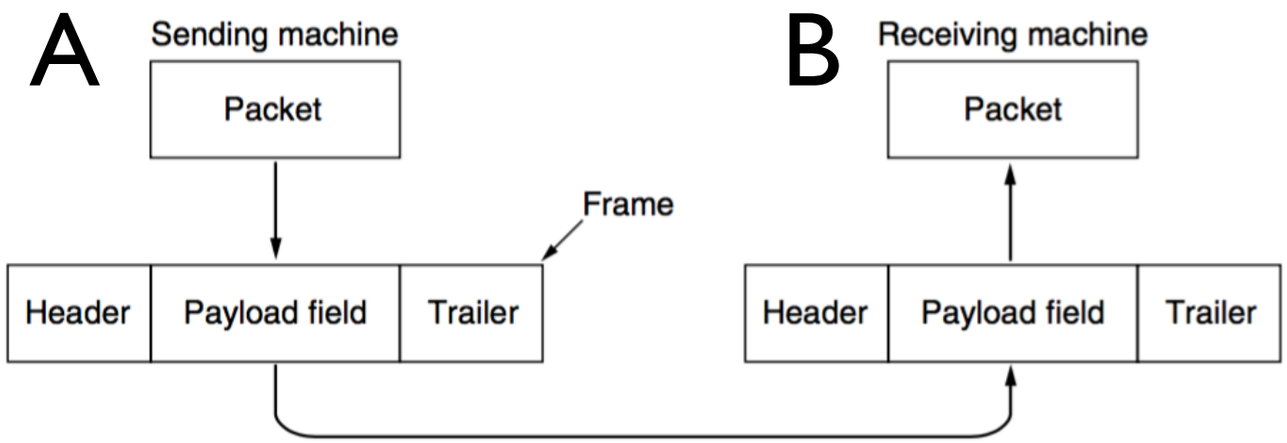
```
typedef enum {data, ack, nak} frame_kind;          /* frame_kind definition */

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

Simplex Stop-and-Wait Protocol




```
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);          /* send it on its way */
    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 */
  }
}
```

Simplex Stop-and-Wait Protocol



Failure scenario

1. A sends frame to B
2. B receives the frame
3. B sends ack to A
4. ack gets lost in transmission
5. A times out
6. A sends the same data again
7. B receives the same frame again ⟶ error

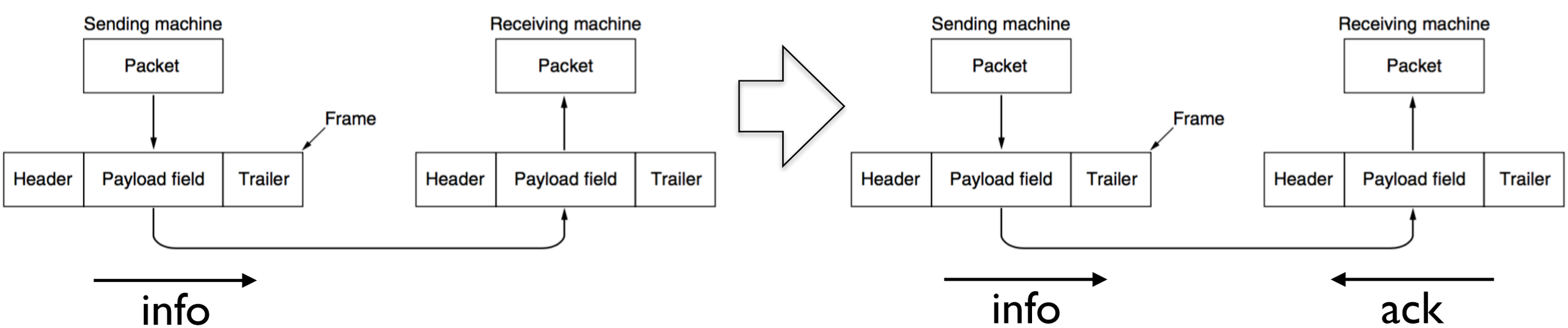
```
typedef struct {                          /* frames are transported in this layer */
  frame_kind kind;                        /* what kind of frame is it? */
  seq_nr seq;                             /* sequence number */
  seq_nr ack;                             /* acknowledgement number */
  packet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

ARQ (Automatic Repeat reQuest)
PAR (Positive Acknowledgement with Retransmission)

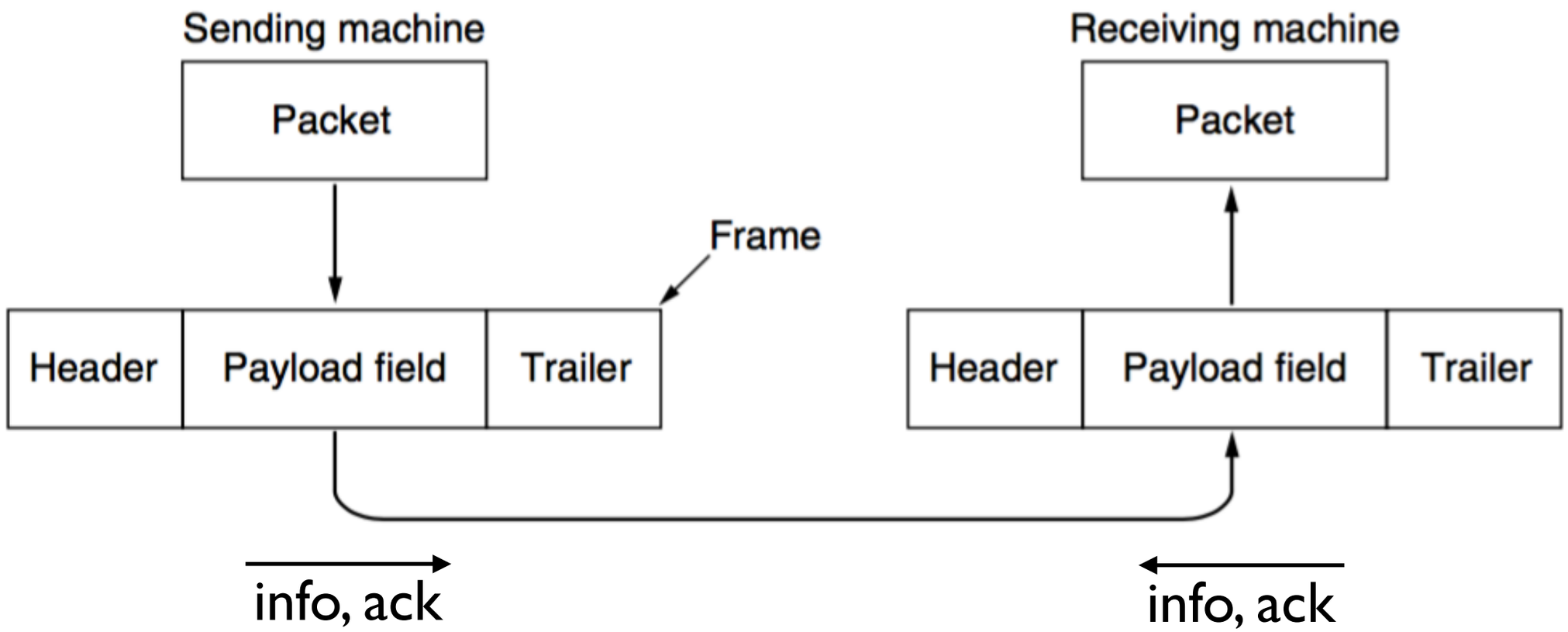
Simplex Stop-and-Wait Protocol with Error

Assumptions

1. ~~Both the transmitting and receiving network layers are always ready~~
2. ~~The communication channel between the data link layers never damages or loses frames~~
3. Data are transmitted in one direction only
4. Processing time can be ignored



Simplex Stop-and-Wait Protocol with Error



Piggybacking

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

Simplex Stop-and-Wait Protocol with Error

```
#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;                              /* buffer for an outbound frame */
  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);        /* increment next_frame_to_send */
      }
    }
  }
}
```

Simplex Stop-and-Wait Protocol with Error

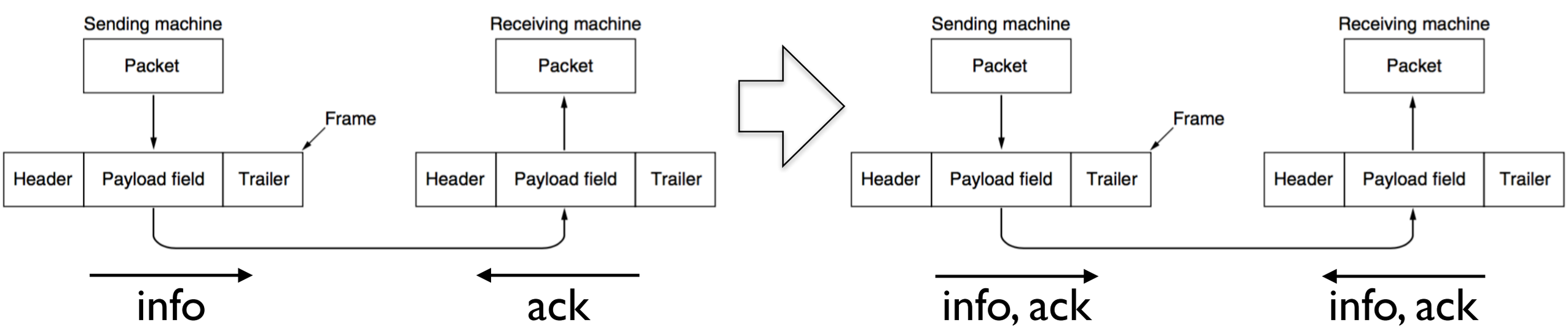
```
void receiver3(void) {
  seq_nr frame_expected;
  frame r, s;
  event_type event;
  frame_expected = 0;
  while (true) {
    wait_for_event(&event);            /* possibilities: frame arrival, cksum err */
    if (event == frame_arrival) {      /* a valid frame has arrived */
      from_physical_layer(&r);         /* 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);           /* increment frame_expected */
      }
      s.ack = 1 – frame_expected;      /* tell which frame is being acked */
      to_physical_layer(&s);           /* send acknowledgement */
    }
  }
}
```

```
typedef struct {                                  /* frames are transported in this layer */
  frame_kind kind;                                /* what kind of frame is it? */
  seq_nr seq;    0,1,0,1,0,1,…                    /* sequence number */
  seq_nr ack;    0,1,0,1,0,1,…                    /* acknowledgement number */
  packet info;                                    /* the network layer packet */
} frame;
```

One Bit Sliding Window Protocol (Duplex Stop-and-Wait)

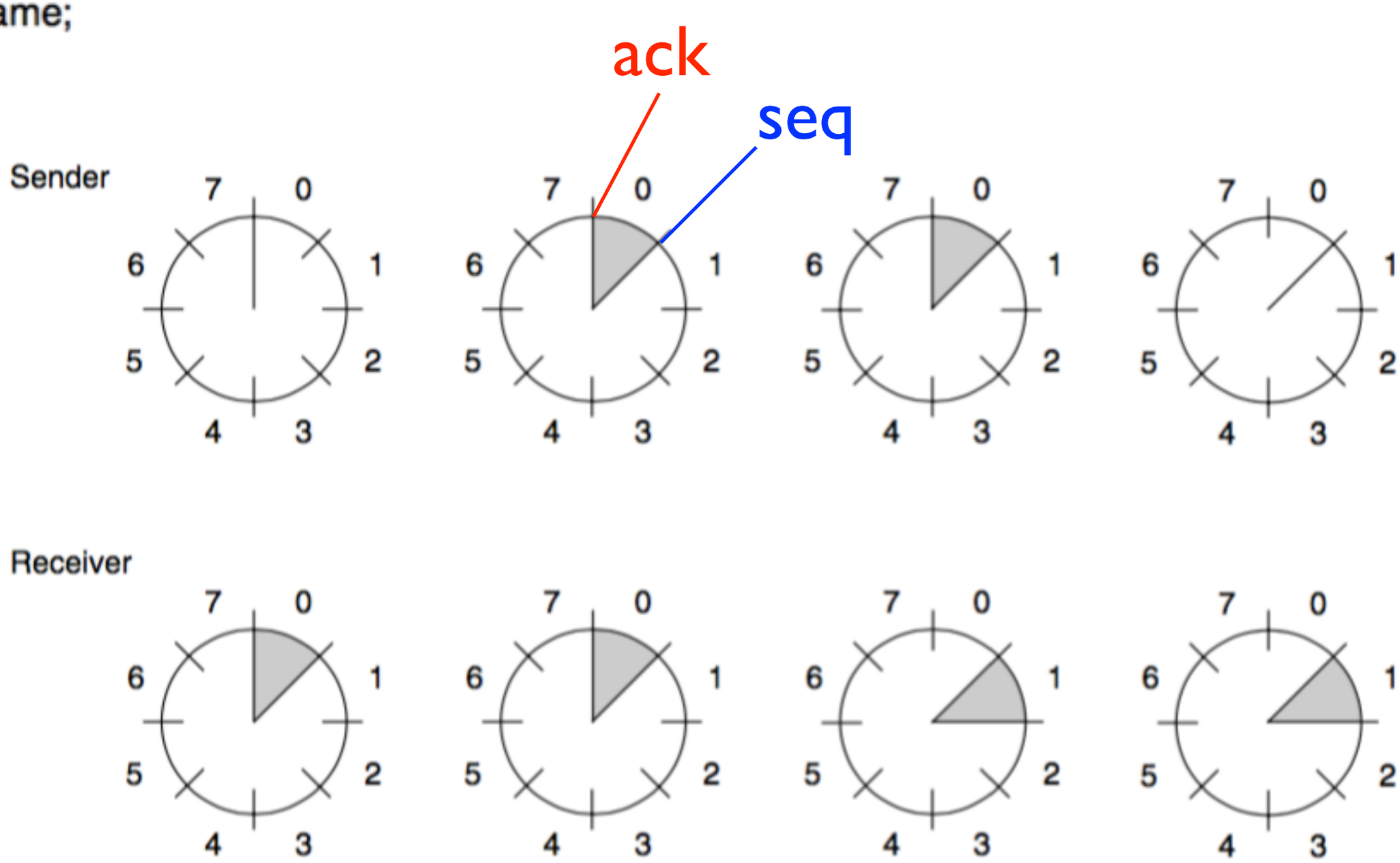
Assumptions

1. ~~Both the transmitting and receiving network layers are always ready~~
2. ~~The communication channel between the data link layers never damages or loses frames~~
3. ~~Data are transmitted in one direction only~~
4. Processing time can be ignored

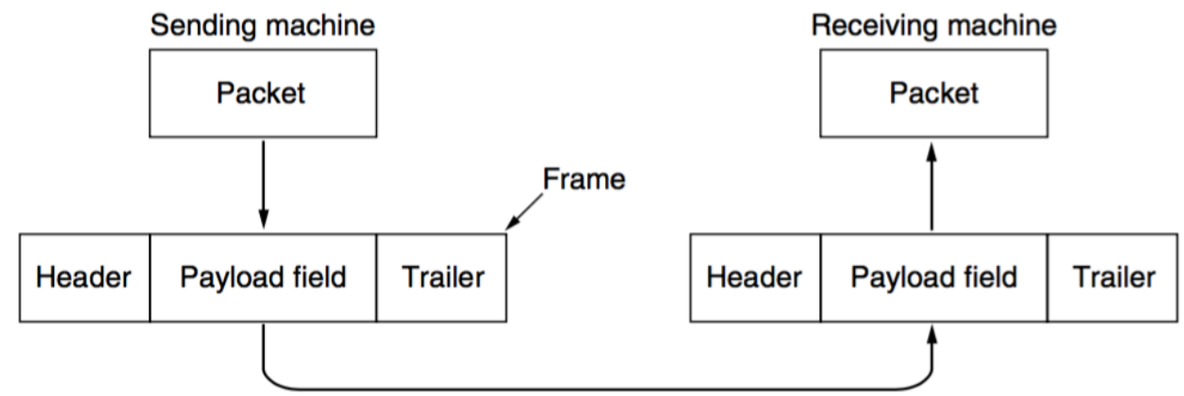


One Bit Sliding Window Protocol

```
typedef struct {                                  /* frames are transported in this layer */
  frame_kind kind;                                /* what kind of frame is it? */
  seq_nr seq;    0,1,2,3,4,5,6,7,0,…              /* sequence number */
  seq_nr ack;    0,1,2,3,4,5,6,7,0,…              /* acknowledgement number */
  packet info;                                    /* the network layer packet */
} frame;
```

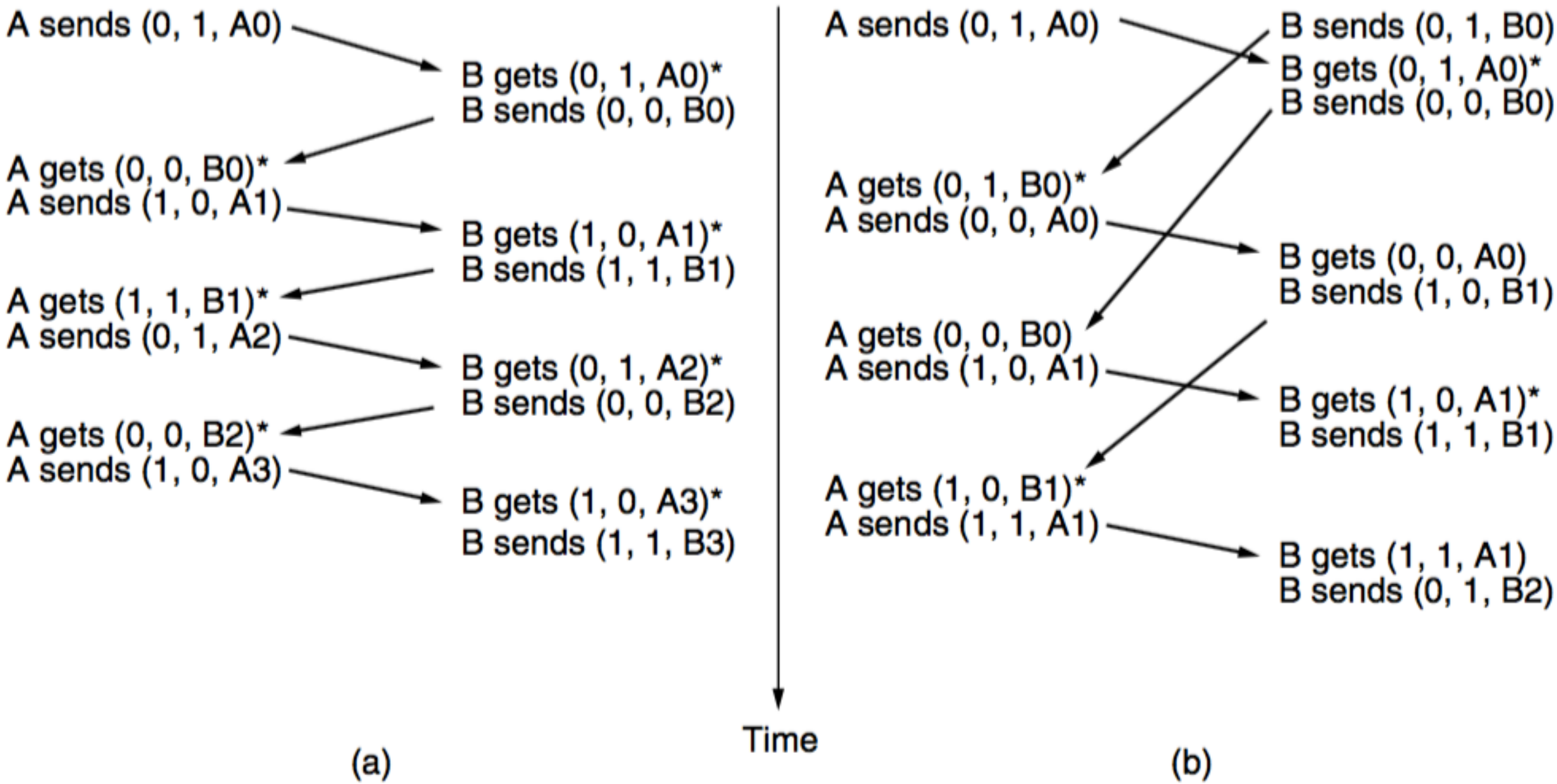


One Bit Sliding Window Protocol




```
#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;                           /* buffer for frames */
  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);            /* increment seq number */
      }
      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 */
  }
}
```

One Bit Sliding Window Protocol

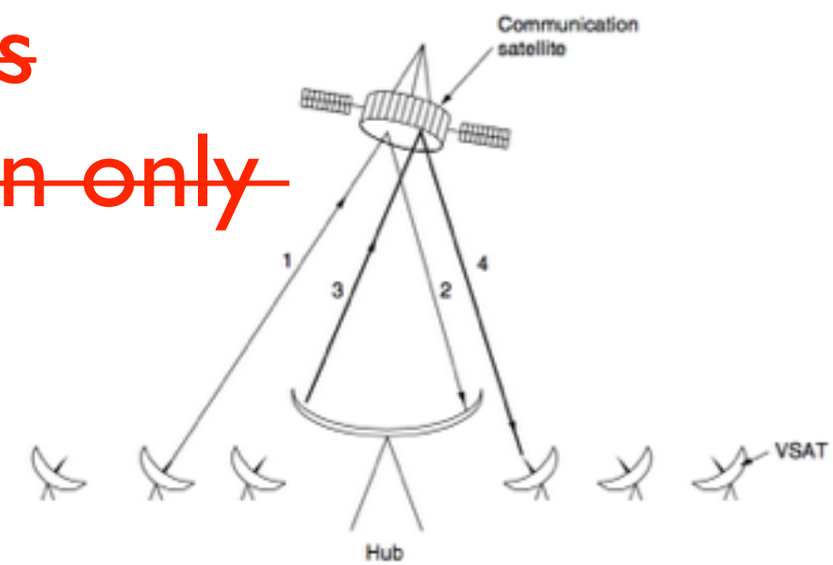


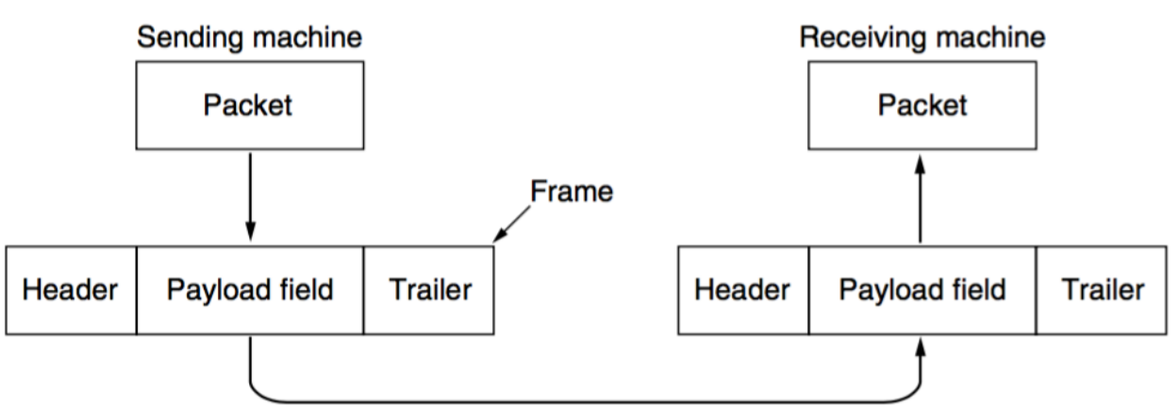
The same frame may be sent two or three times

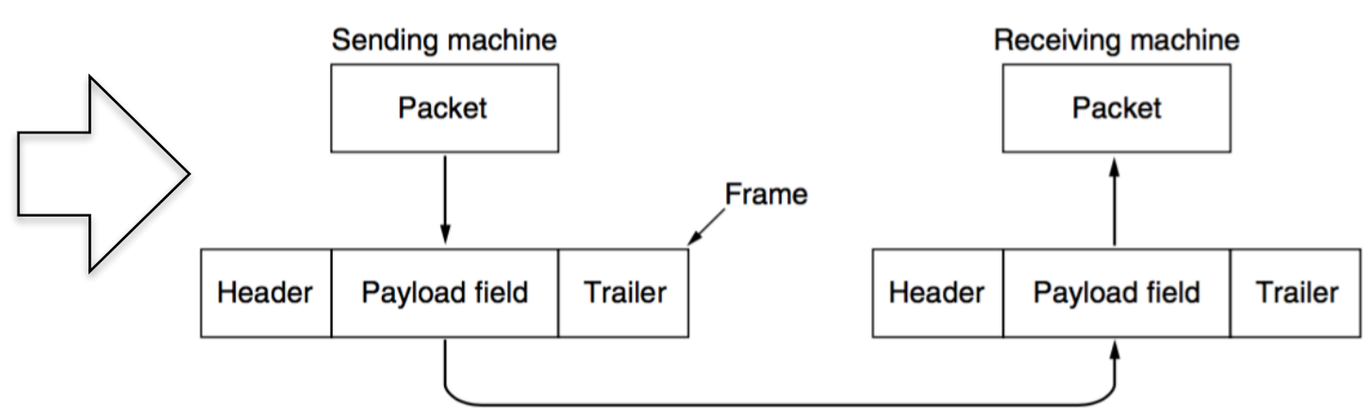
Go-back-N protocol

Assumptions

1. ~~Both the transmitting and receiving network layers are always ready~~
2. ~~The communication channel between the data link layers never damages or loses frames~~
3. ~~Data are transmitted in one direction only~~
4. ~~Processing time can be ignored~~





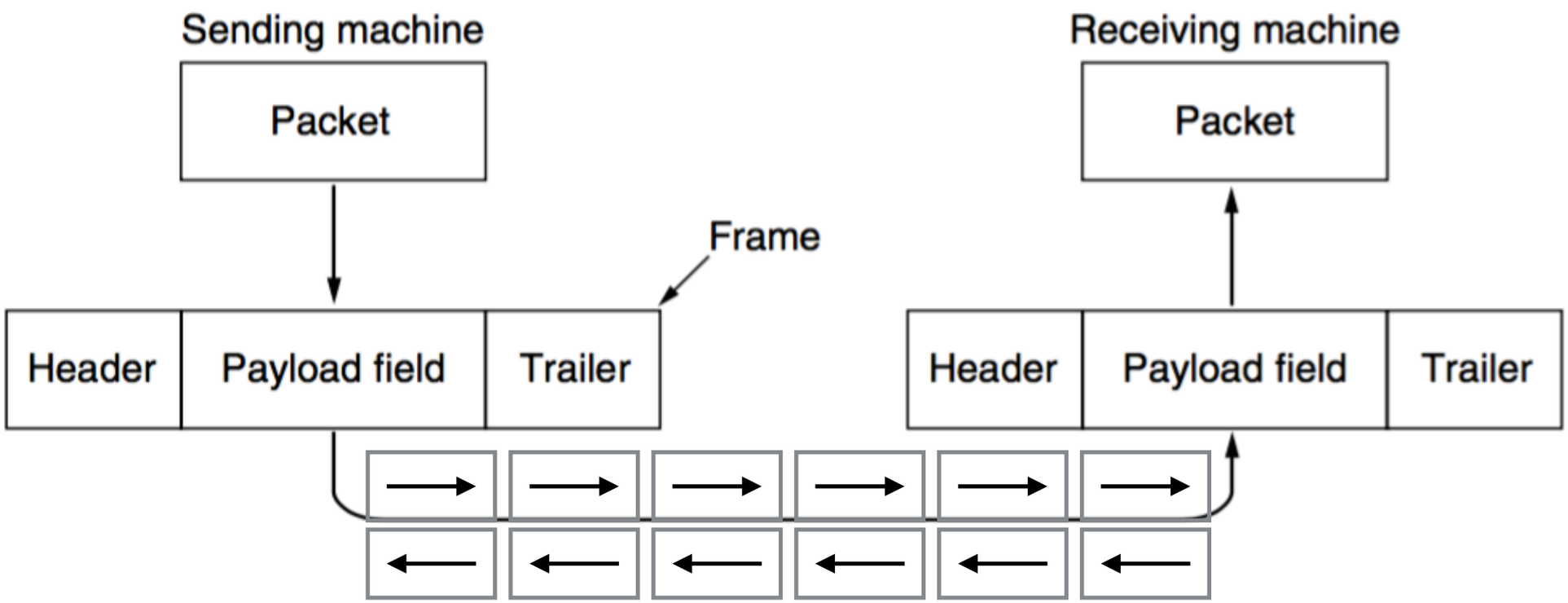


info, ack info, ack

info, ack info, ack info, ack

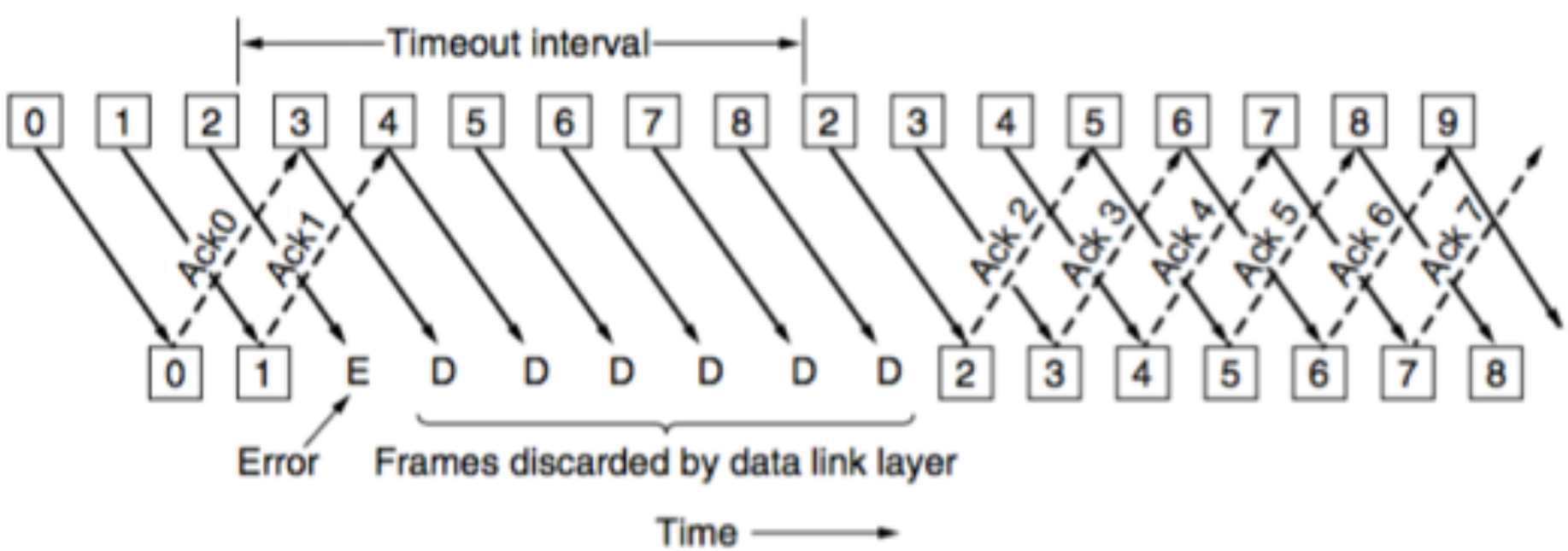
Go-back-N protocol

BD (Bandwidth Delay product) = (Bandwidth) x (One way transit time)



w=2BD+1

$$\text{link utilization} \leq \frac{w}{1+2BD}$$



Go-back-N protocol

```
#define MAX_SEQ 7
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;                                                    /* buffer for frames */
  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;                                                    /* buffer for frames */
  packet buffer[MAX_SEQ+1];                                   /* buffers for the outbound stream */
  seq_nr nbuffered;                                           /* number of 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 */
```

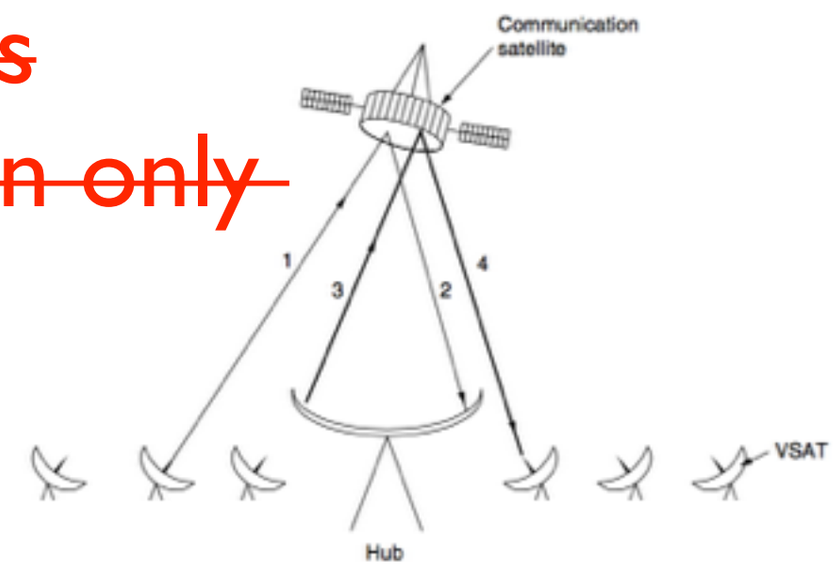
Go-back-N protocol

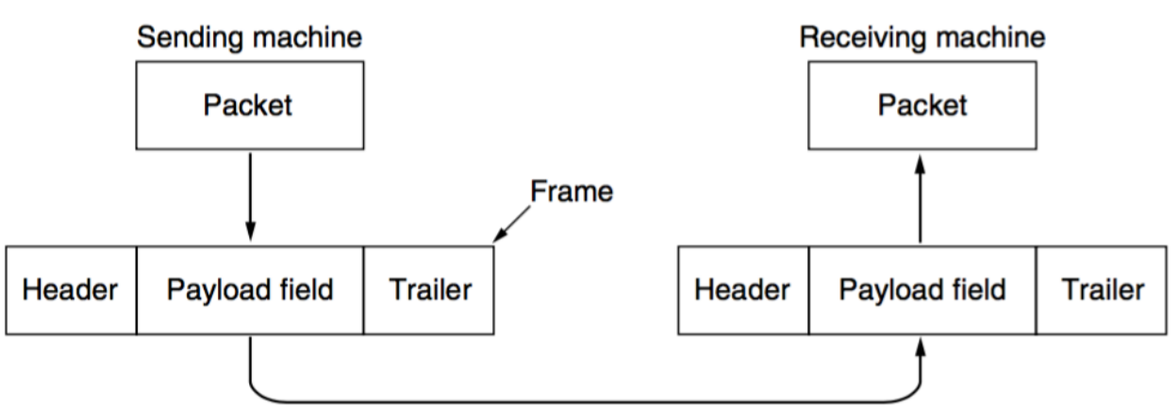
```
while (true) {
  wait_for_event(&event);                                  /* four possibilities: see event type above */
  switch(event) {
    case network_layer_ready:                              /* the network layer has a packet to send */
      from_network_layer(&buffer[next_frame_to_send]);     /* 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 */
      }
      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 frame */
        inc(next_frame_to_send);                           /* prepare to send the next one */
      }
    }
    if (nbuffered < MAX_SEQ)
      enable_network_layer();
    else
      disable_network_layer();
  }
}
}
```

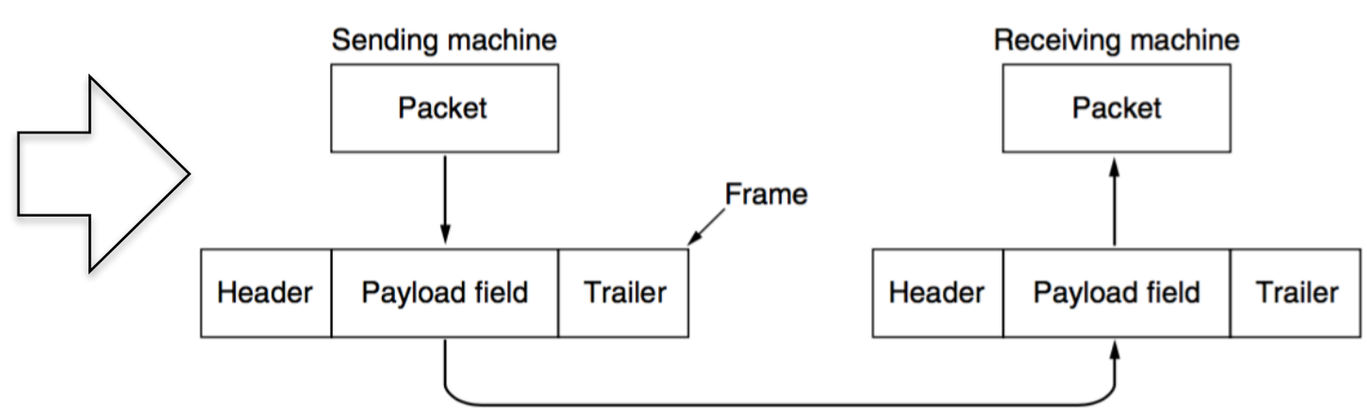
Selective Repeat protocol

Assumptions

1. ~~Both the transmitting and receiving network layers are always ready~~
2. ~~The communication channel between the data link layers never damages or loses frames~~
3. ~~Data are transmitted in one direction only~~
4. ~~Processing time can be ignored~~



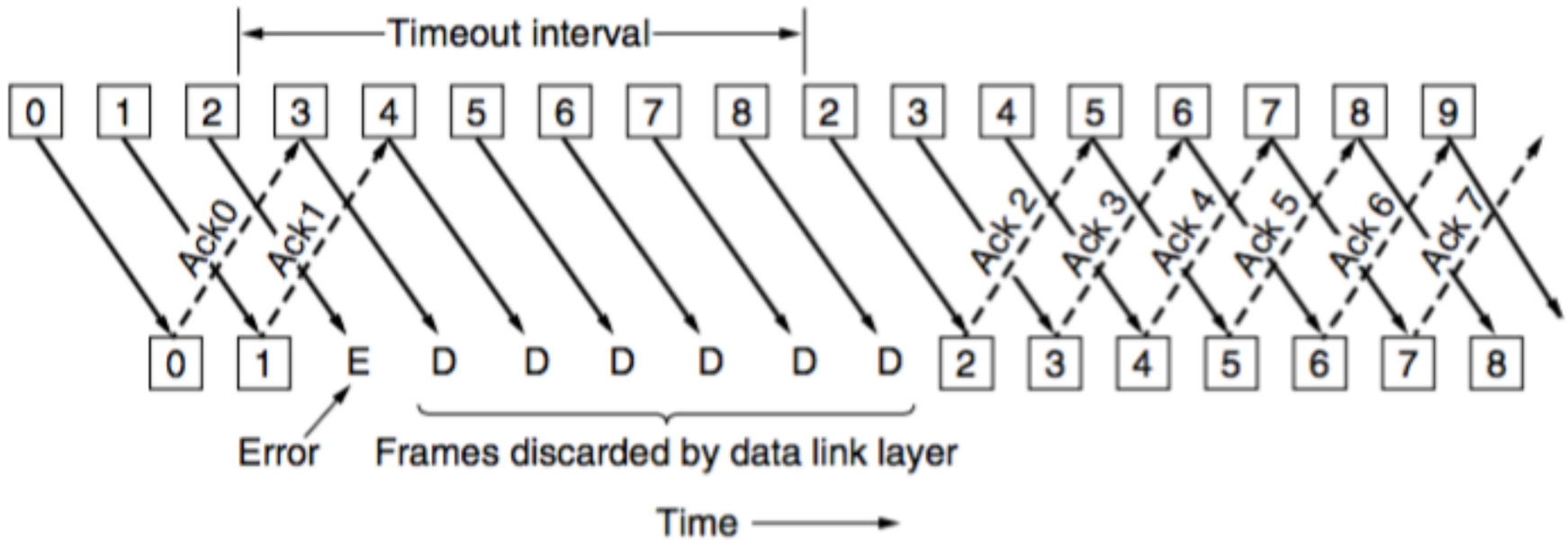




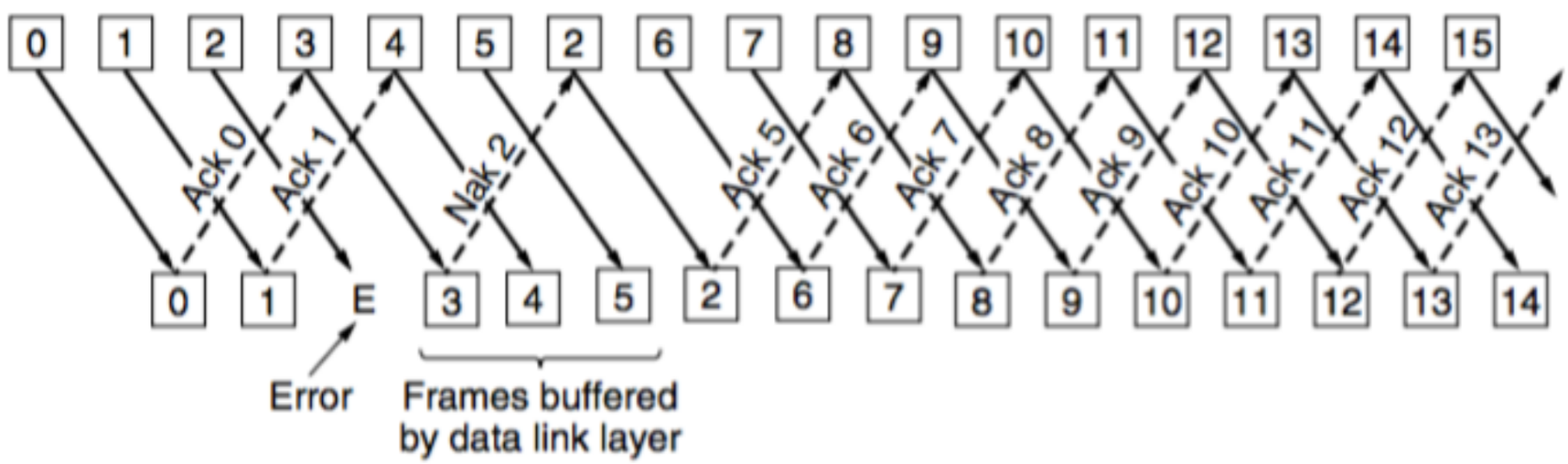
info, ack info, ack

info, ack info, ack info, ack

Selective Repeat protocol



Go-back-N protocol



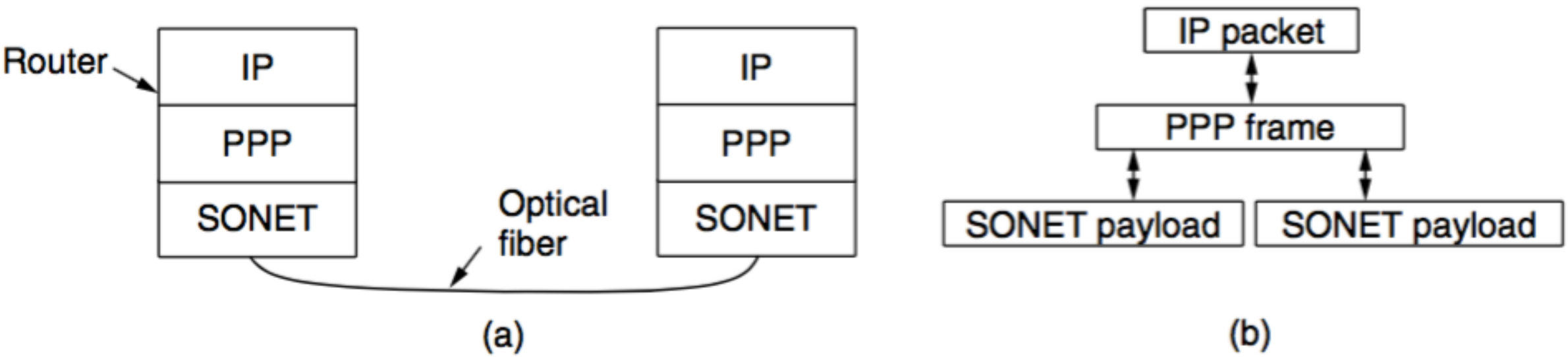
Selective Repeat protocol

```
#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) {
  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[]) {
  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;                                              /* buffer for frames */
  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;
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)) {
          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();
  }
}
```

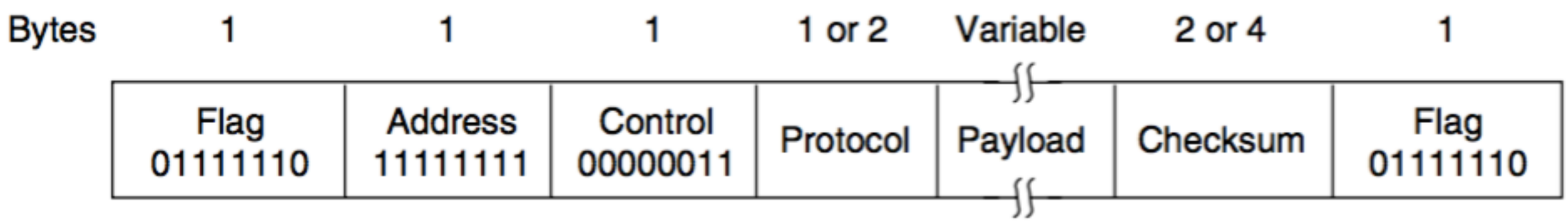
Point-to-Point Protocols (PPP) over SONET



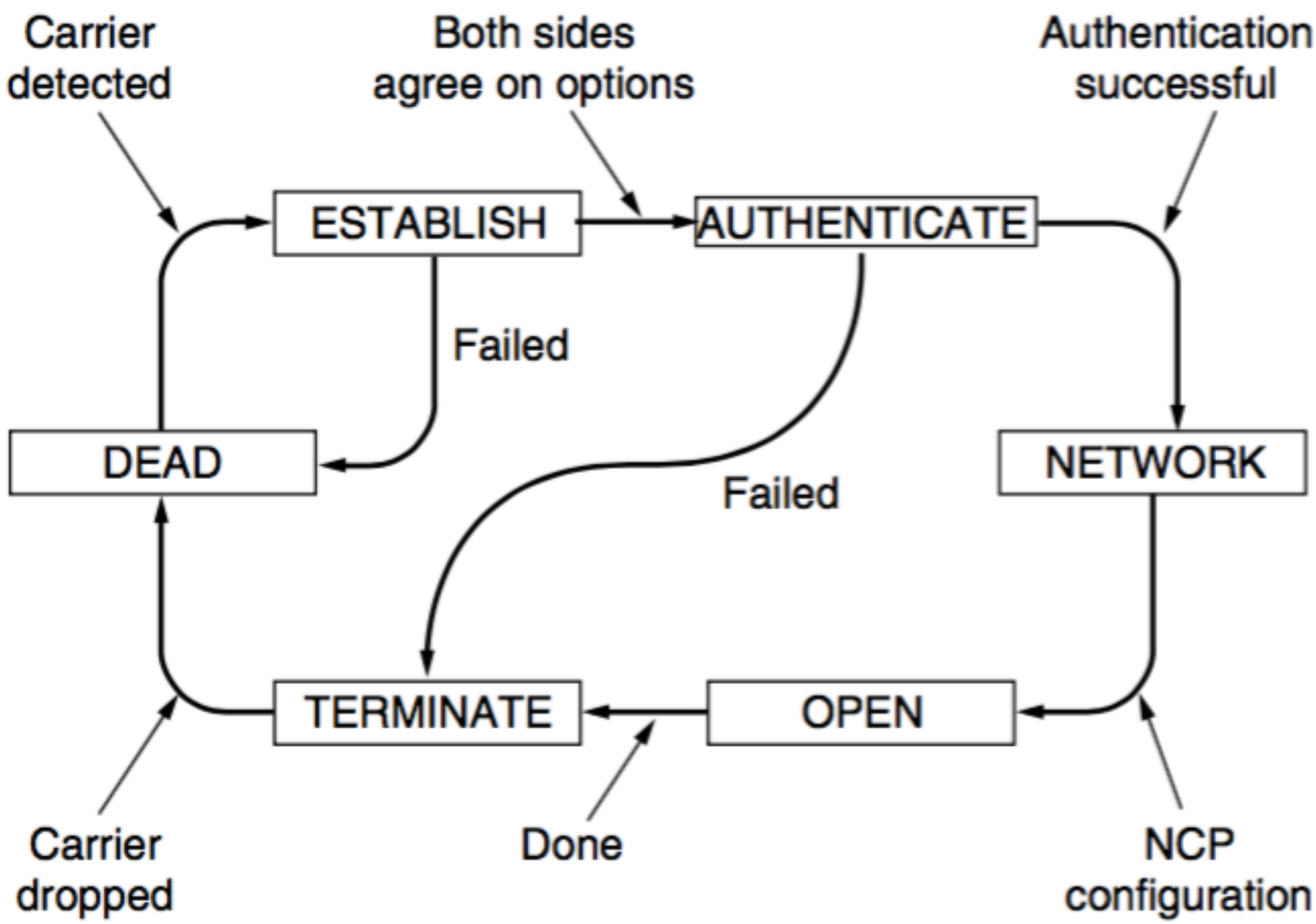
Packet over SONET. (a) A protocol stack. (b) Frame relationships.

Features of PPP

1. A framing method that unambiguously delineates the end of one frame and the start of the next one. The frame format also handles error detection.
2. A link control protocol (LCP) for bringing lines up, testing them, negotiating options, and bringing them down again gracefully when they are no longer needed.
3. A way to negotiate network-layer options in a way that is independent of the network layer protocol to be used. The method chosen is to have a different NCP (Network Control Protocol) for each network layer supported.

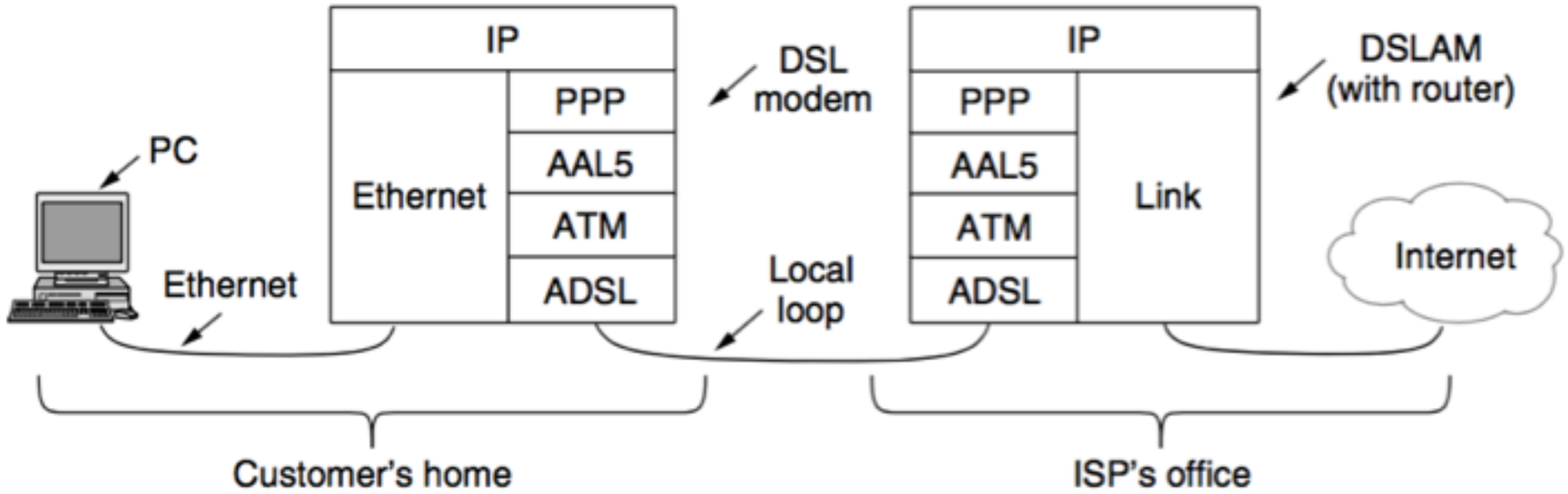


PPP over SONET

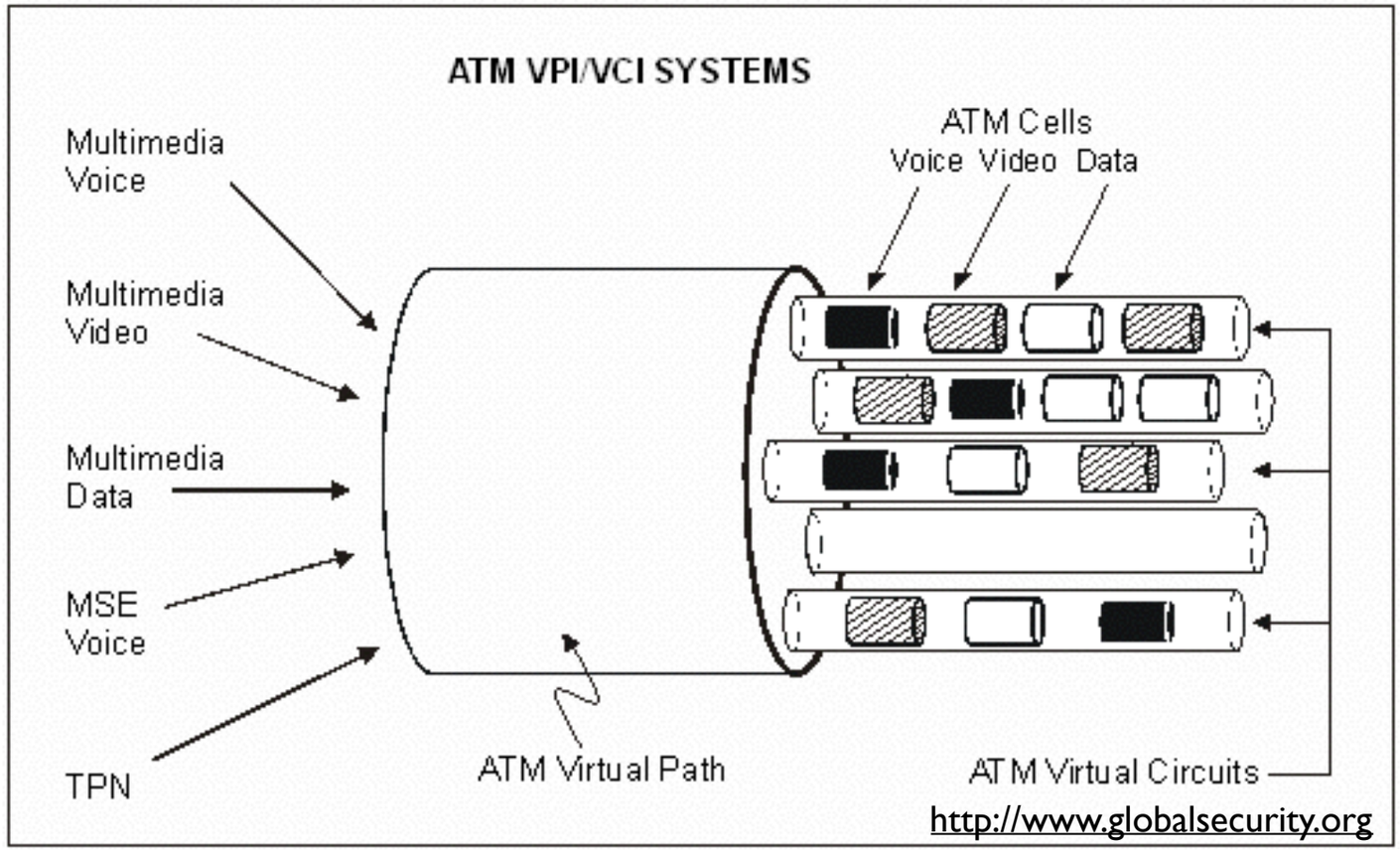


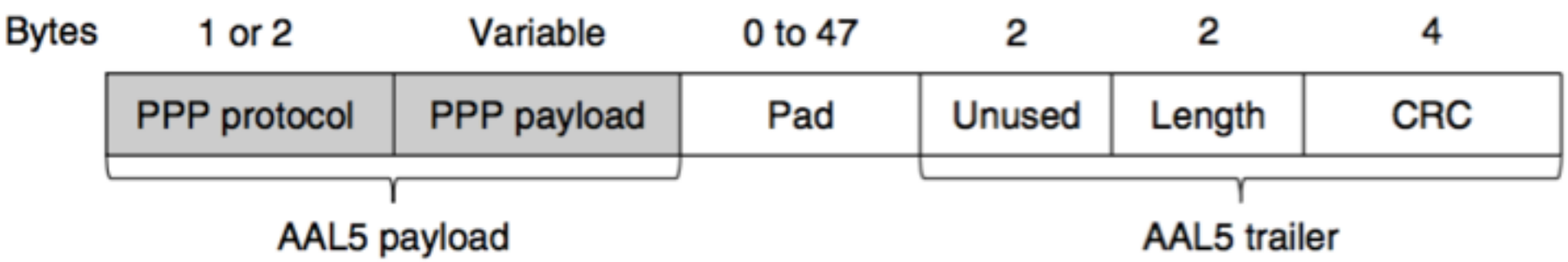
State diagram for bringing a PPP link up and down.

PPP over ADSL



Asynchronous Transfer Mode (ATM)
AAL5 (ATM Adaptation Layer 5)





講義日程（1Q）

		授業計画		課題
04/05	第１回	計算機ネットワークの基本概念 ハードウェア・ソフトウェア，参照モデル	１章	ネットワークの種類と参照モデルを理解し プロトコル階層と各層の設計課題
04/12	第２回	物理層１ 有線伝送と無線伝送	２章	物理チャネルの特性を理解し データ通信の理論的基礎を理解
04/19	第３回	物理層２ デジタル変調と多重化	２章	ベースバンド伝送と通過帯域伝送， 電話網，携帯電話システムを説明できる
04/26	第４回	データリンク層１ 誤りの検出・訂正	３章	誤りの検出・訂正のしくみを理解し 検出・訂正符号の計算ができる
05/10	第５回	データリンク層２ データリンク・プロトコル	３章	データリンク・プロトコルの種類， 各プロトコルを定量的に評価できる
05/17	第６回	メディア・アクセス副層１ ブロードキャスト・チャネル	４章	多重アクセス・プロトコルを理解し データ・レートを計算できる
05/24	第７回	メディア・アクセス副層２ 無線 LAN，Bluetooth, RFID	４章	個別のプロトコル・スタックを理解し データリンク層スイッチングを理解
05/31	第８回	理解度確認総合演習 (中間試験) 第１回から第７回までの内容の演習形式による確認		第１回から第７回までの理解度確認と 到達度自己評価