Kompiuterių tinklai - duomenų perdavimo kanalo sluoksnis

Trečia paskaita (3.1 - 3.5 skyriai),
http://computernetworks5e.org/chap03.html

lekt. Vytautas Jančauskas

Duomenų kadrai

Duomenų perdavimo kanalo sluoksnio uždaviniai

Į pagrindines duomenų perdavimo kanalo sluoksnio funkcijas įeina:

1. Suteikti tinklo sluoksniui patogų intefeisą duomenims perduoti.
2. Apdoroti perdavimo klaidas.
3. Užtikrinti kad greitas siuntėjas neperkrautų lėtų priėmėjų.

Duomenų pakeitai gaunami iš tinklo sluoksnio yra sudedami į tam tikras duomenų struktūras kurias vadinsime duomenų kadrais.

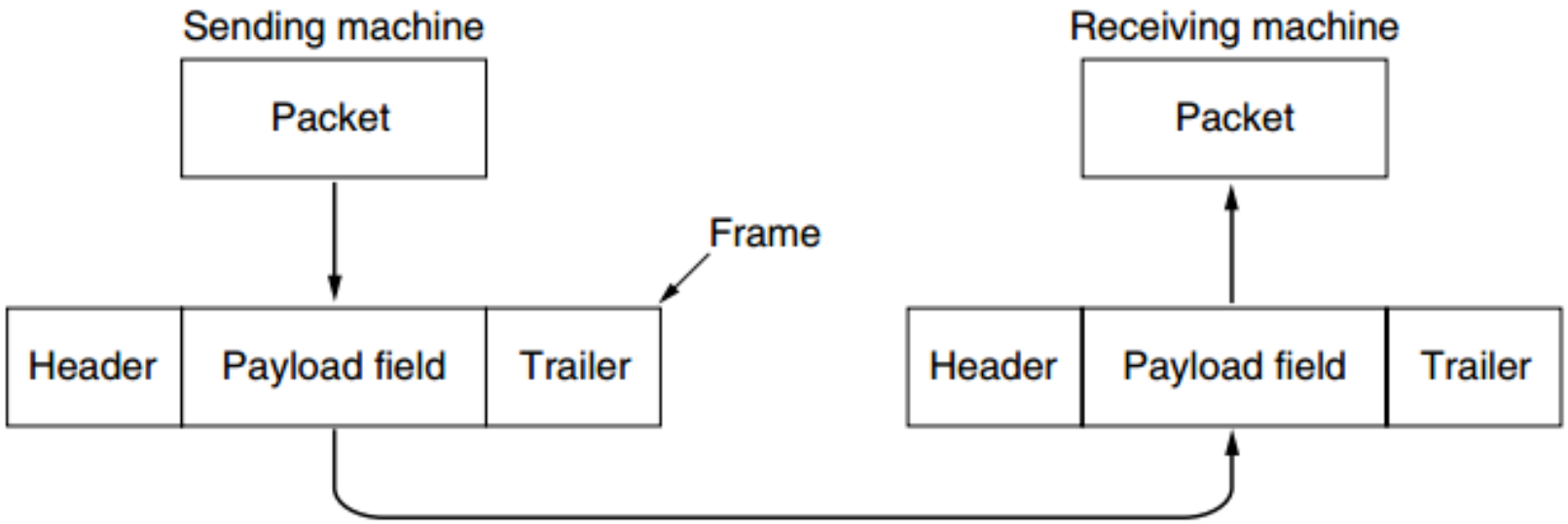


Figure 3-1. Relationship between packets and frames.

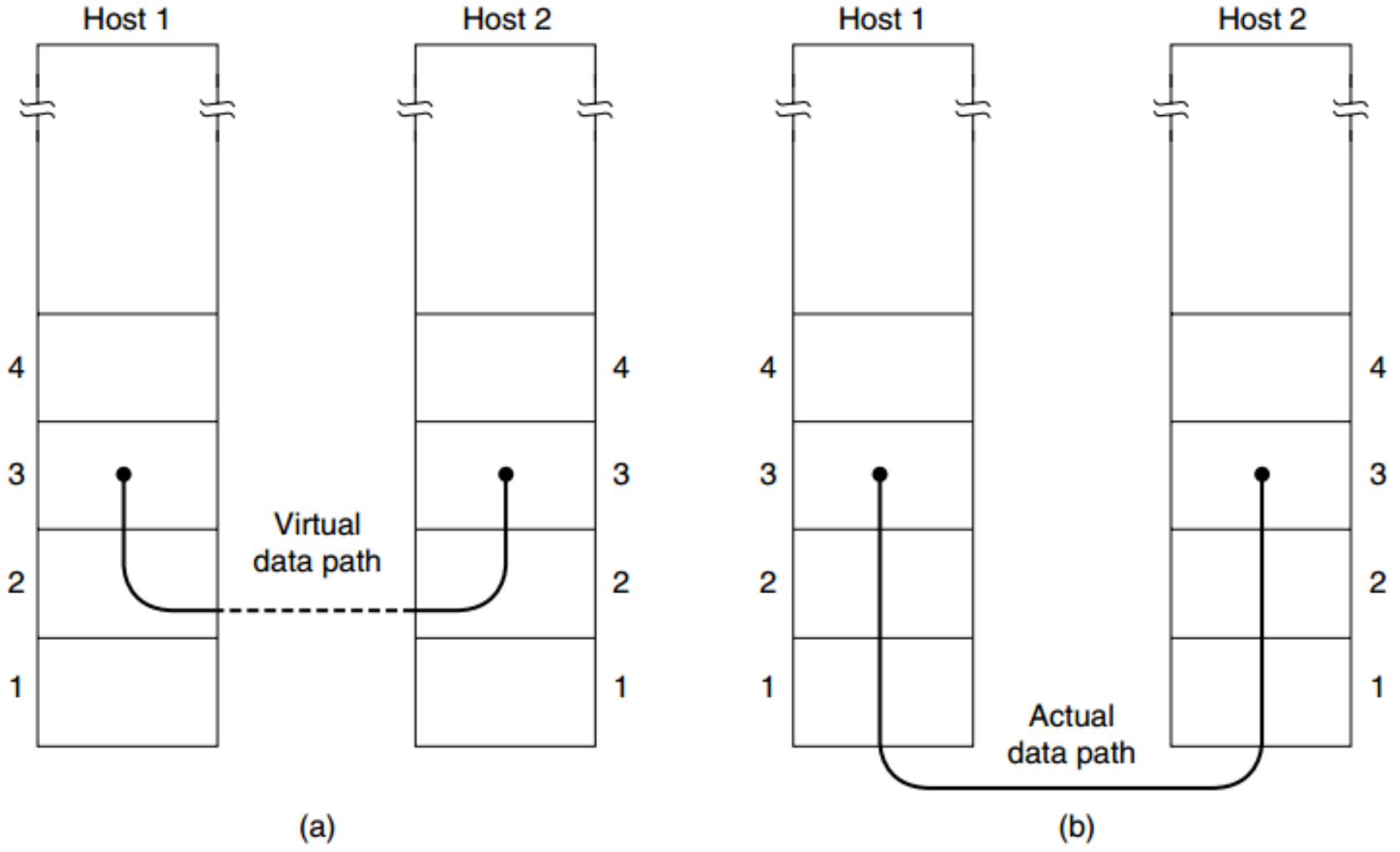


Figure 3-2. (a) Virtual communication. (b) Actual communication.

Duomenų perdavimo kanalo sluoksnio tekiamos paslaugos

Išskirsime tris pagrindinius paslaugų tipus kurias gali teikti duomenų perdavimo kanalo sluoksnis.

1. Paslaugos be sujungimo ir be patvirtinimo.
2. Paslaugos be sujungimo su patvirtinimu.
3. Paslaugos su sujungimu ir patvirtinimu.

Ethernet yra protokolo pavyzdys teikiantis paslaugą 1, **802.11** teikia paslaugą 2.

Duomenų dalinimas kadrais

Dalinant duomenis į kadrus reikia nustatyti kadro pradžią ir pabaigą bitų sraute. Tam gali būti naudojami šie būdai:

1. Baitų skaičiavimas.
2. Specialūs baitai naudojant **byte stuffing**.
3. Specialūs bitai naudojant **bit stuffing**.
4. Fizinio sluoksnio kodavimo pažeidimai.

Baitų skaičiavimo atveju tiesiog prieš duomenis skiriama kažkiek baitų duomenų dydžiui įrašyti. Fizinio sluoksnio kodavimo pažeidimų atvejui naudojamos bitų kombinacijos kurios negalimos pvz. esant 4B/5B kodavimui.

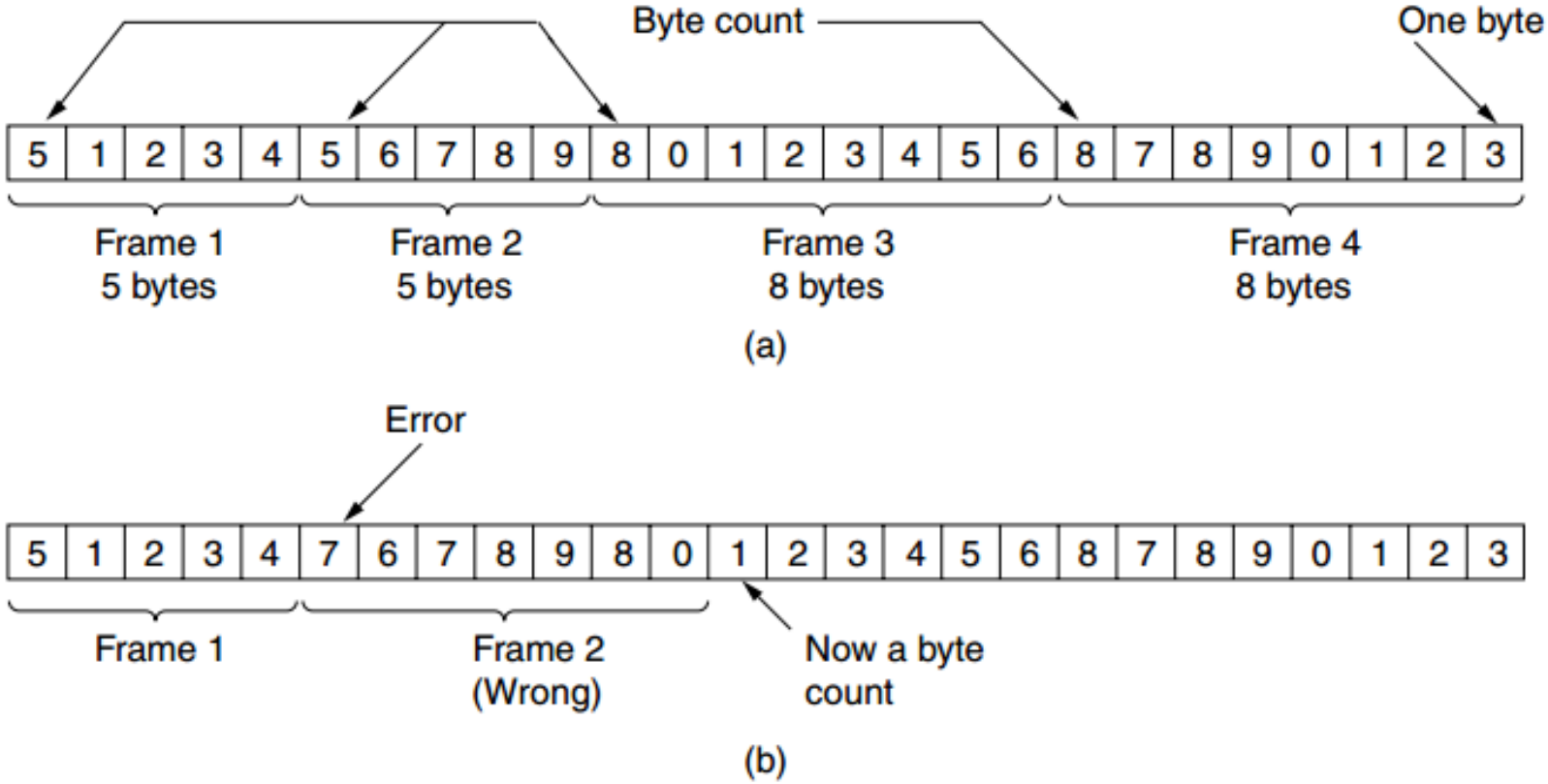


Figure 3-3. A byte stream. (a) Without errors. (b) With one error.

Dalinimas į kadrus naudojant specialų FLAG baitą

- Naudojamas specialus FLAG baitas žymėti kadro pradžiai ir pabaigai.
- Kas atsitinka jeigu baitas su tokia pačia reikšme kaip FLAG yra duomenyse?
- Naudojamas specialus ESC baitas kuris įterpiamas prieš FLAG baitą.
- Jeigu duomenyse sutinkamas ESC baitas prieš jį taip pat įterpiamas ESC baitas.
- Gavus duomenis ESC baitai panaikinami.

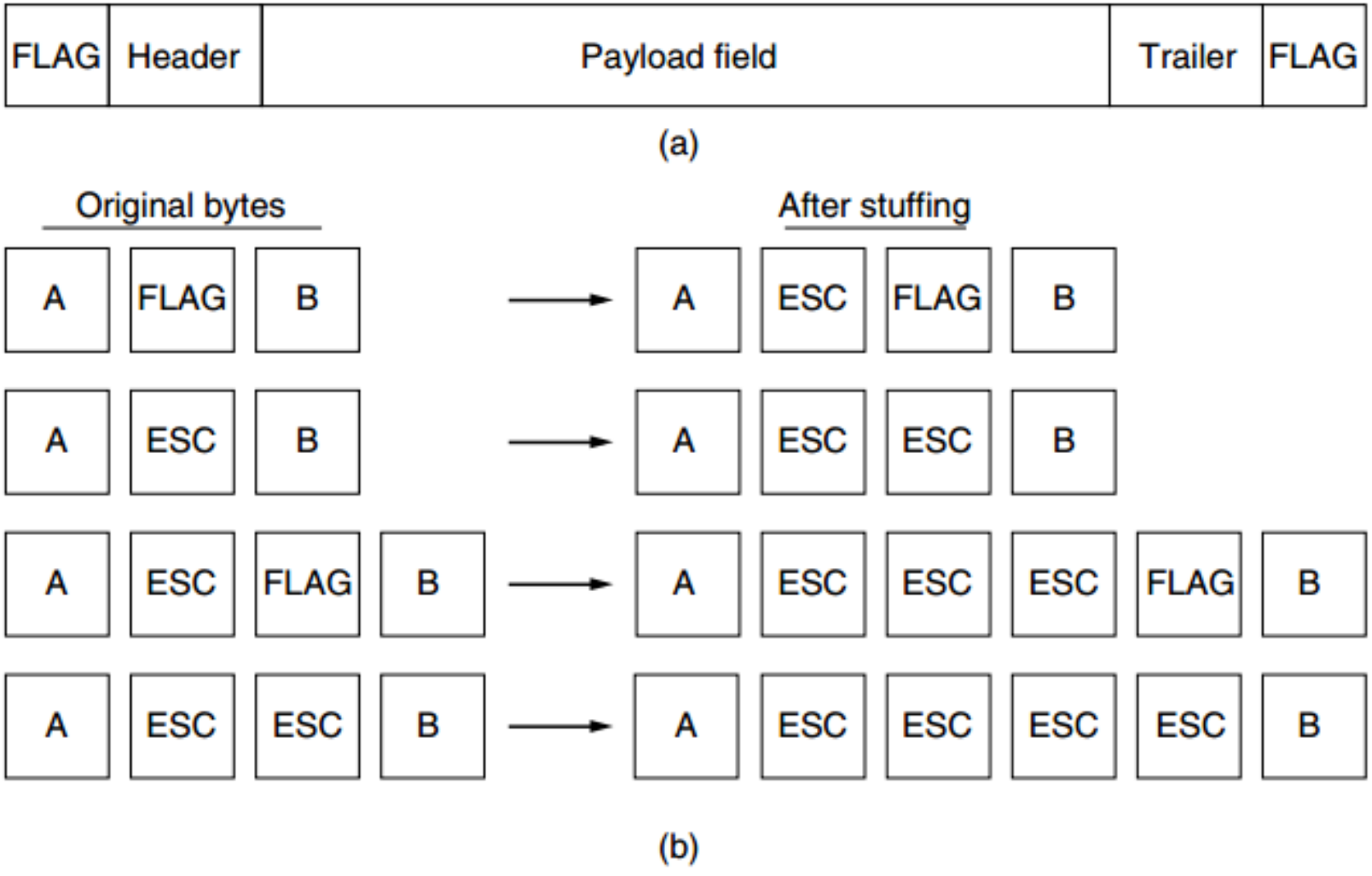


Figure 3-4. (a) A frame delimited by flag bytes. (b) Four examples of byte sequences before and after byte stuffing.

FLAG bitai

- Visi kadrai pradedami ir baigiami bitų seka 01111110 arba 0x7E.
- Jeigu duomenyse sutinkami 5 vienetai po jų įterpiamas nulis.
- Taip užtikrinama, kad seka 0x7E nebus sutikta duomenyse.
- Gavus duomenis jeigu po 5 vienetų seka nulis, jis yra ištrinamas.
- Taip sutaupoma bitų perduodamuose duomenyse palyginus su ESC baitais.

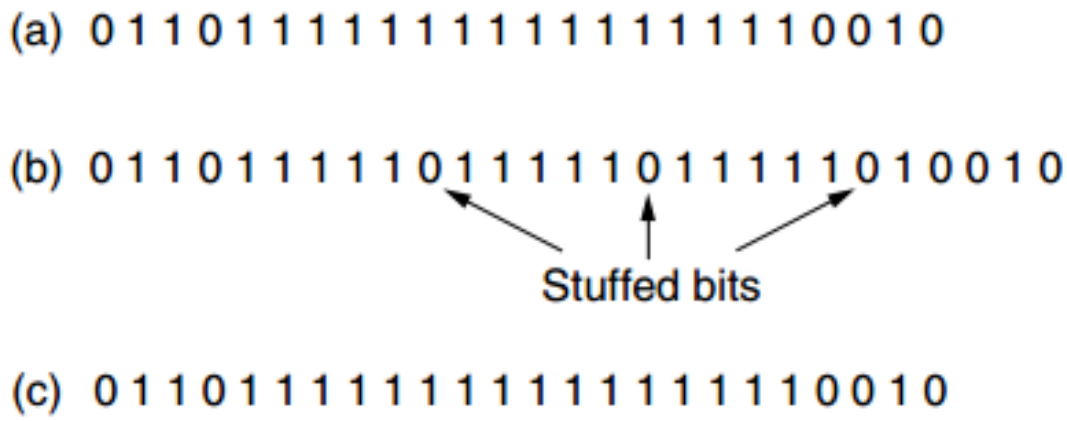


Figure 3-5. Bit stuffing. (a) The original data. (b) The data as they appear on the line. (c) The data as they are stored in the receiver's memory after destuffing.

Klaidų kontrolė

- Kartais reikia užtikrinti, kad duomenys tikrai pasiekė gavėją.
- Tam, paprasčiausiu atveju, naudojamas specialus duomenų kadras kuris nusiunčiamas siuntėjui, gavėjui gavus duomenis.
- Kartais kadrai gali būti visai prarandami. Tokiu atveju gavėjas negalės išsiusti pranešimo apie gautus duomenis.
- Naudojami taimeriai nustatyti laikui, per kurį negavus patvirtinimo duomenys gali būti siunčiami iš naujo.
- Siekiant užtikrinti eiliškumą ir kad tie patys duomenys nebūtų siunčiami du kartus, kadrai numeruojami.

Tekmės kontrolė (**Flow control**)

- Reikia užtikrinti, kad kadrai nebus siunčiami greičiau, nei juos gali priimti gavėjas. Praktikoje taip gali atsitikti dažnai.
- Paprastai to išvengiama palaukiant kol gavėjas praneš, kad gali priimti naujus duomenis.
- Kitas būdas yra įvesti apribojimus kaip greitai siuntėjas gali siųsti duomenis.

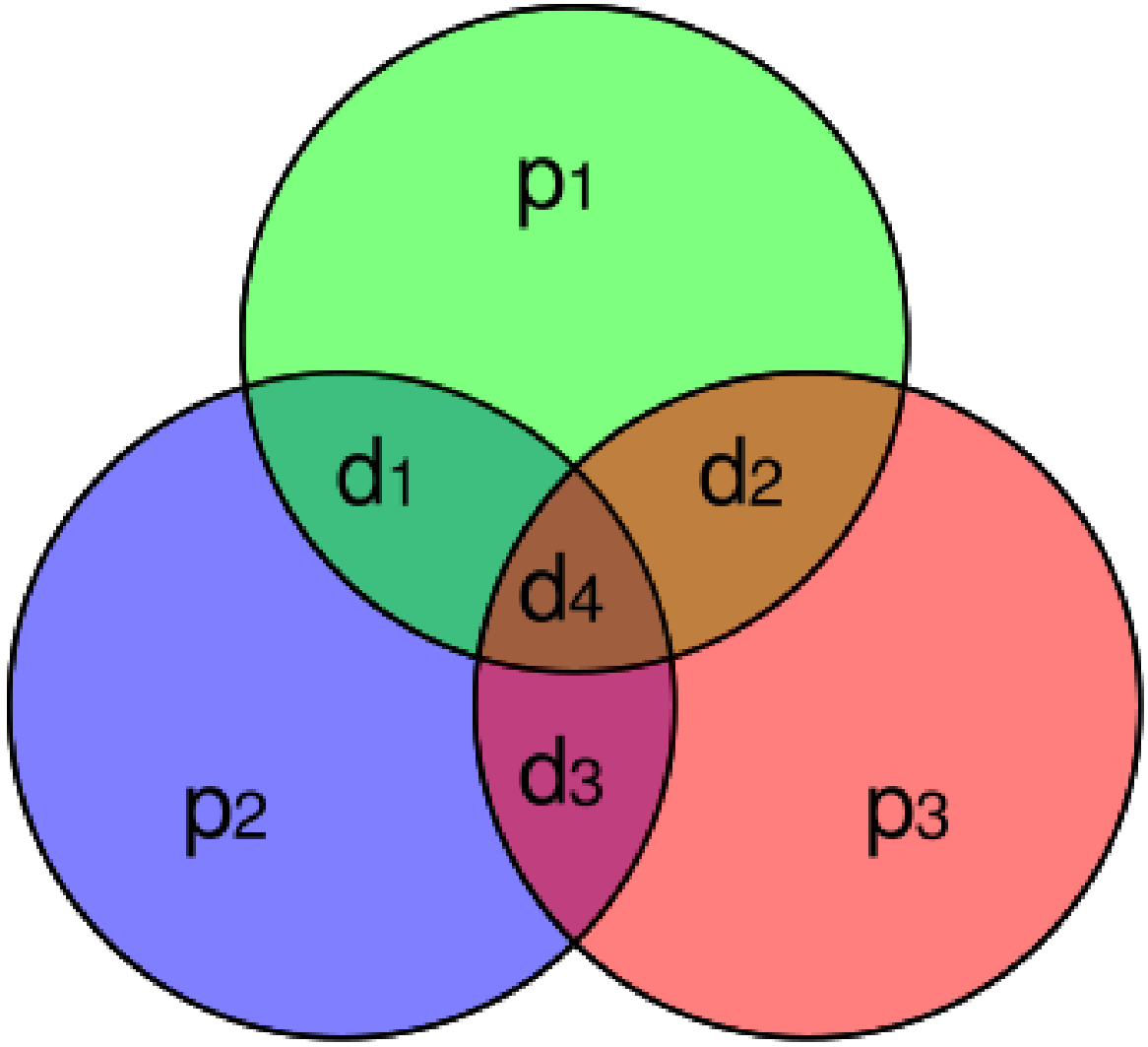
Klaidas taisantys kodai

Aptarsime šiuos klaidas taisančius kodus:

1. Hamming kodai.
2. Dvejetainiai sasūkos kodai.
3. Reed-Solomon kodai.
4. LDPC kodai.

Hamming kodai

- Blokinis kodas.
- Tarp duomenų įterpiami poriškumo skaičiavimo bitai.
- Pavyzdžiui (7, 4) Hamming kodas atrodo taip - $p_1, p_2, d_1, p_3, d_2, d_3, d_4$.
- Tokiu atveju 1011 bus užkoduota kaip 1011011.
- Kaip atstatyti duomenis įvykus vienai klaidai, pvz. 1111011?


p1
d1
d2
d4
d3
p2
p3

Dvejetainiai sąsukos kodai

- Iš nagrinėjamų vienintelis ne blokinis kodas.
- Šiuose koduose įeinančių bitų srautas generuoja išeinančių bitų srautą.
- Šie kodai atkoduojami randant bitų srauta kuriam tikimybė generuoti gautą užkoduotą srautą yra didžiausia.
- Atkodavimui naudojamas Viterbi pasiūlytas algoritmas.

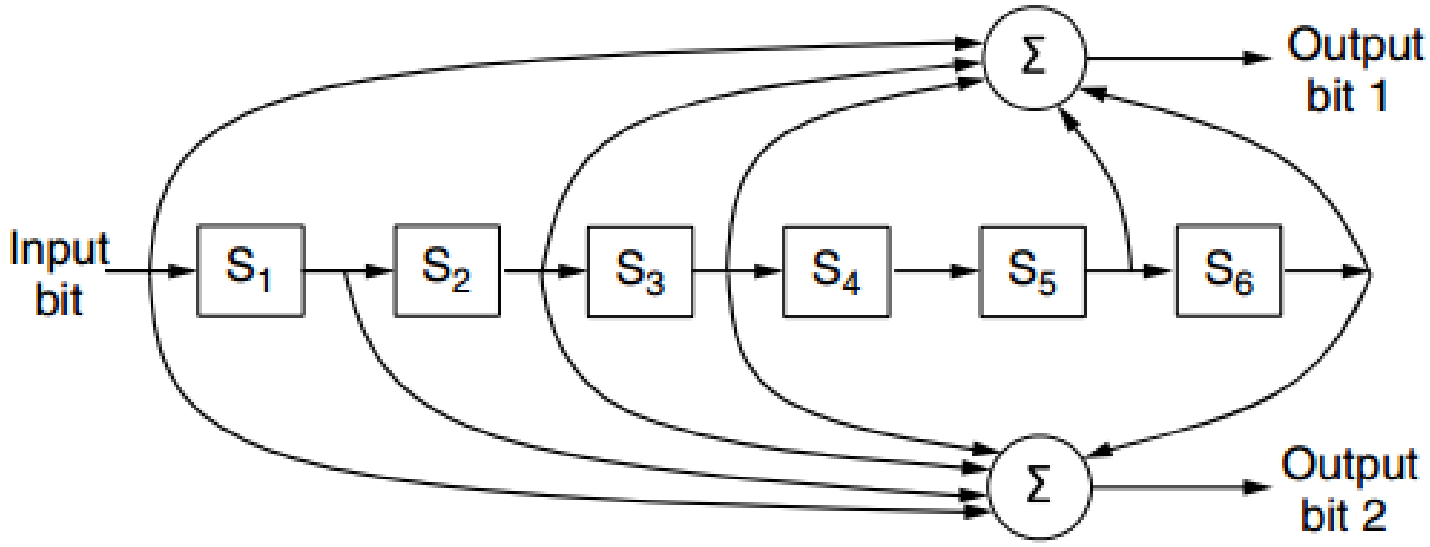


Figure 3-7. The NASA binary convolutional code used in 802.11.

Reed-Solomon kodai

- Tiesiniai blokiniai kodai.
- Naudojasi matematinėmis polinomų savybėmis.
- Plačiai naudojami praktikoje dėl gerų klaidų taisymo savybių.
- Naudojami DSL, palydovinėje komunikacijoje, CD, DVD, Blue-Ray standartuose.
- Dažnai naudojami kombinuojant su sąsukos kodais.

LDPC kodai

- LDPC - Low-Density Parity Check
- Tiesiniai blokiniai kodai sukurti 1962.
- Tinka dideliems blokų dydžiams, praktikoje veikia geriau nei dauguma kitų kodų.
- Naudojami video transliavime, 10 Gbps Ethernet ir 802.11n standartuose.

Klaidas aptinkantys kodai (I)

- Jeigu klaidos kanale yra retos, galima naudoti klaidas aptinkančius kodus vietoje taisančių. Tokiu atveju, įvykus klaidai paketas būtų tiesiog persiunčiamas iš naujo.
- Aptarsime kelis tokius kodus:
  1. Poriškumas.
  2. Kontrolinės sumos.
  3. Ciklinė perteklinė kontrolė (**CRC - Cyclic Redundancy Check**).

Poriškumas

- Pats paprasčiausias būdas klaidoms aptikti.
- Prie duomenų pridedamas poriškumo bitas.
- Suskaičiuojame vienetų skaičių duomenyse. Jeigu jis porinis tai poriškumo bito reikšmė vienas, priešingu atveju nulis.
- Tinka kai klaidų tankis nedidelis.
- Gali būti naudojamas klaidų sekoms aptikti naudojant **interleaving** metodą.

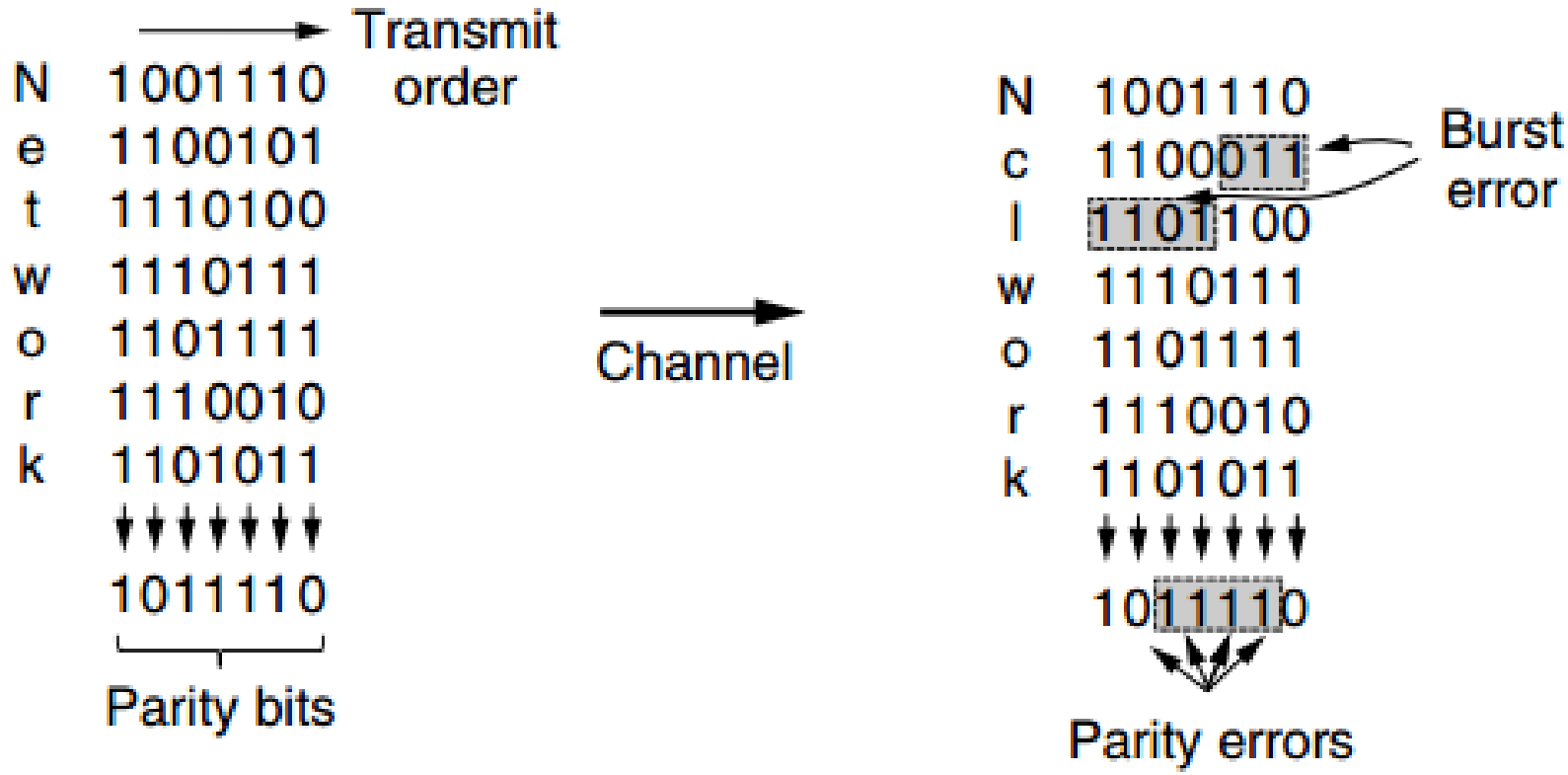


Figure 3-8. Interleaving of parity bits to detect a burst error.

Kontrolinės sumos

- Paprasčiausiu atveju galime sudėti duomenų baitus, paimti rezultatą moduliu 2^8 ir naudoti tai kaip kontrolinę sumą. Pasikeitus duomenims, pasikeis ir suma.
- Interneto kontrolinę sumą, naudojamą visuose IP paketuose, sudaro 16 bitų. Jie gaunami padalinus duomenis 16 bitų žodžiais ir juos sudėjus. Tada sumoje vientai pakeičiami nuliais o nuliai vienetais.
- Fletcher'io kontrolinės sumos algoritmas pateikiamas žemiau.

```
1.  uint16_t Fletcher16( uint8_t* data, int count )
2.  {
3.     uint16_t sum1 = 0;
4.     uint16_t sum2 = 0;
5.     int index;
6.
7.     for( index = 0; index < count; ++index )
8.     {
9.        sum1 = (sum1 + data[index]) % 255;
10.       sum2 = (sum2 + sum1) % 255;
11.    }
12.
13.    return (sum2 << 8) | sum1;
14. }
```

Ciklinė perteklinė kontrolė

- Polinominis kodas.
- Siuntėjas ir gavėjas susitaria dėl generuojančio polinomo $G(x)$.
  1. Tarkime r yra polinomo $G(x)$ laipsnis. Pridedame r nulinių bitų prie žinutės galo taip kad jos ilgis būtų $m + r$ ir atitiktų polinomą $x^r M(x)$.
  2. Daliname bitų eilutę atitinkančią $x^r M(x)$ iš bitų eilutės atitinkančios $G(x)$ naudojant dalybą moduliu 2.
  3. Atimame liekaną iš bitų eilutės atitinkančios $x^r M(x)$ naudojant atimtį moduliu 2. Rezultatas yra pranešimas su kontroline suma kurį ir siųsime. Šį polinomą vadinsime $T(x)$.

Ethernet naudojamas $G(x)$:

$$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x^1 + 1 \qquad (1)$$

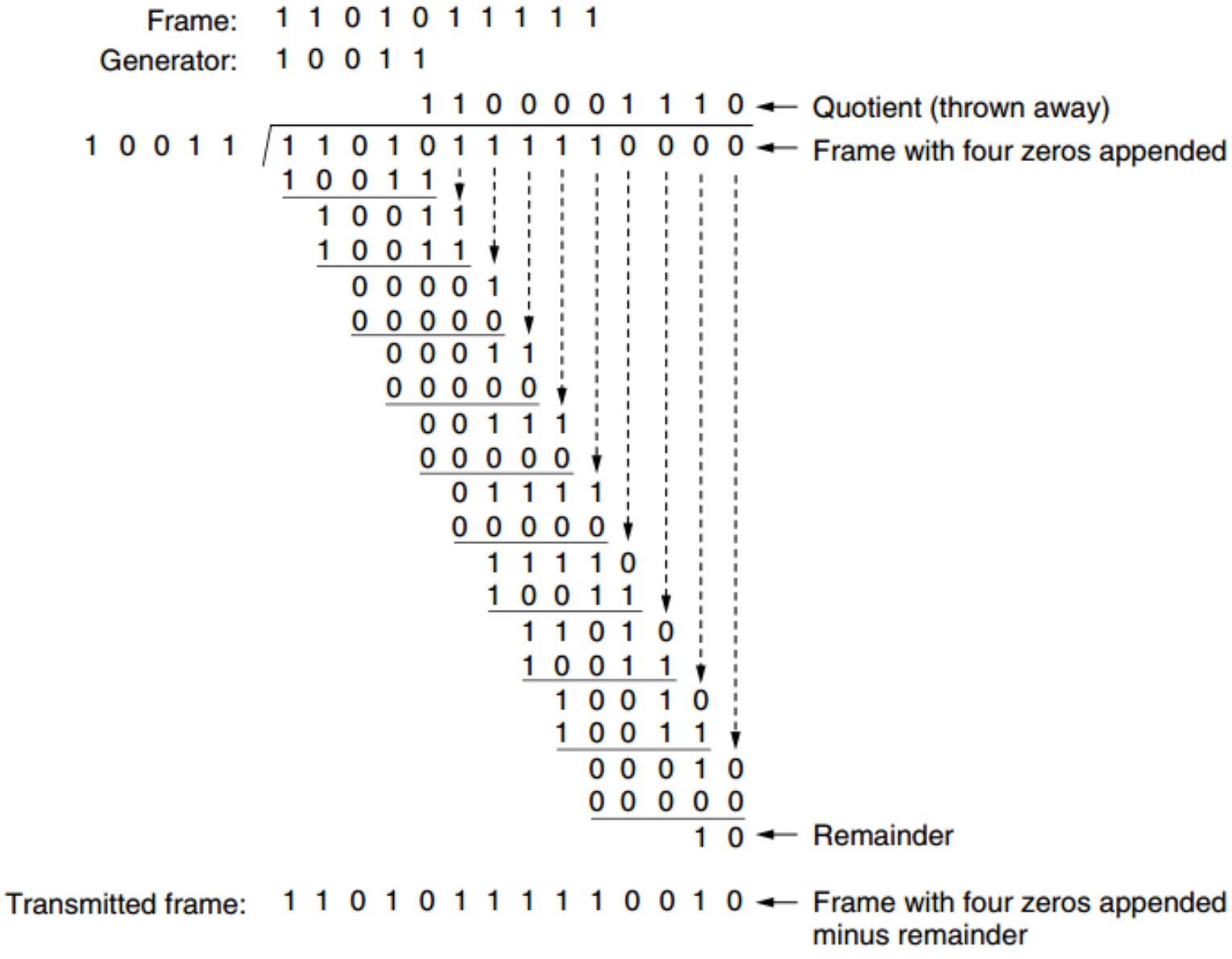


Figure 3-9. Example calculation of the CRC.

Duomenų perdavimo kanalų protokolai

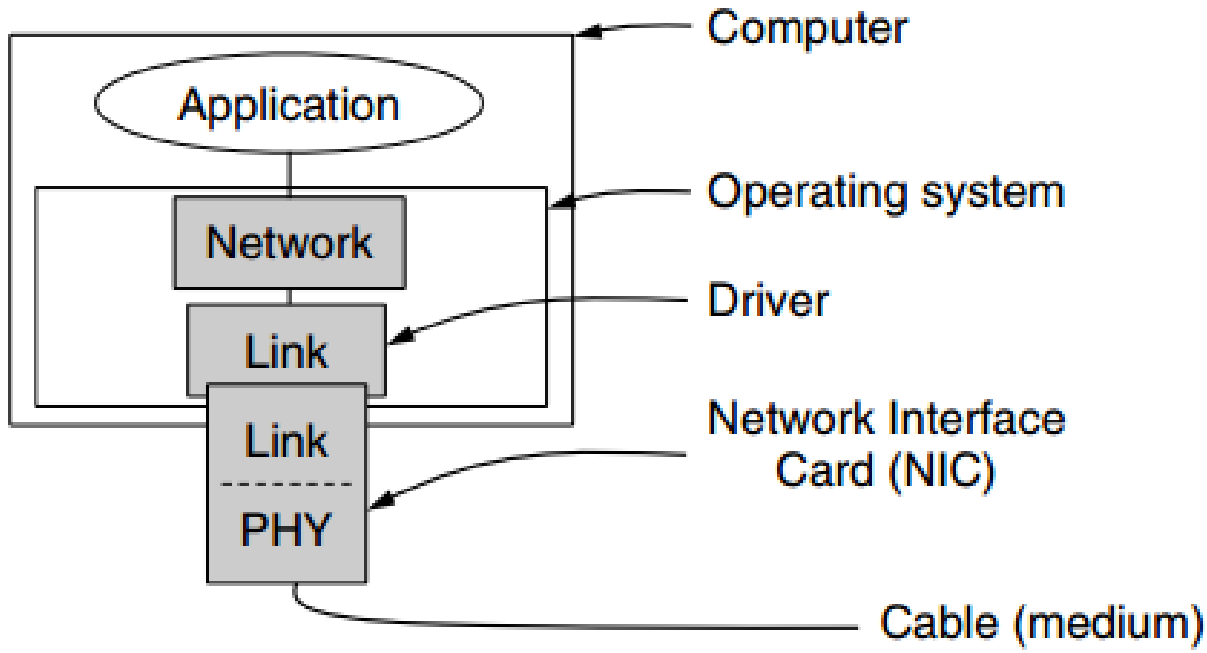


Figure 3-10. Implementation of the physical, data link, and network layers.

Duomenų perdavimo kanalų protokolai

- Toliau darysime tokias prielaidas:
  1. Fizinis sluoksnis, duomenų kanalo sluoksnis ir tinklo sluoksnis yra atskiri procesai bendraujantys žinutėmis.
  2. Kompiuteris A nori siųsti duomenis kompiuteriui B naudodamas patikimą paslaugą su sujungimu. Tarsime, kad A visada turi duomenų kuriuos reiktų siųsti.
  3. Taip pat tarsime kad abu kompiuteriai veikia be sutrikimų (nelūžta).
- Duomenų perdavimo kanalo sluoksnis gavęs paketą iš tinklo sluoksnio interpretuoja jį kaip grynus duomenis, prideda savo informacijos ir perduoda fiziniam sluoksniui išsiųsti.

```
#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;


/* 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
```

Figure 3-11. Some definitions needed in the protocols to follow. These definitions are located in the file *protocol.h*.

Vienkryptis protokolas nr. 1 (utopinis)

Siuntėjas:

1. Paima duomenis iš tinklo sluoksnio.
2. Sudeda juos į duomenų kadro duomenų struktūrą.
3. Perduoda juos fiziniam sluoksniui išsiųsti.

Gavėjas:

1. Blokuojasi laukdamas kol įvyks įvykis “frame_arrival”.
2. Nuskaitomi duomenys iš fizinio sluoksnio.
3. Reikiama duomenų dalis perduodama tinklo sluoksniui.

Praktikoje toks algoritmas negalėtų būti naudojamas. Jo trūkumai - neužtikrinama klaidų ir tekmės kontrolė.

```
/* 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 */
  }
}
```

Figure 3-12. A utopian simplex protocol.

Vienkryptis protokolas Nr. 2 - **stop-and-wait** kanalui be triukšmo

- Skiriasi nuo prieš tai aptarto protokolo tik tuo, kad nusiunčiamas patvirtinimas, kad duomenų kadras gautas.
- Protokolas yra vienkryptis (**simplex**) tačiau, kadangi kanalo siuntėjui ir gavėjui vienu metu nereikia, gali būti naudojamas tas pats. Taigi, užtenka kad kanalas leistų duomenis siųsti abiem kryptimis, nebūtinai tuo pačiu metu.
- Siuntėjas turi palaukti kol gaus atsakymą iš gavėjo, prieš siųsdamas naujus duomenis.

```
/* 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 */
  }
}
```

Figure 3-13. A simplex stop-and-wait protocol.

Vienkryptis protokolas Nr. 3 - **stop-and-wait** kanalui su triukšmu

- Kanale su triukšmu gali atsitikti taip, kad duomenų kadras bus pamestas.
- Atrodytų tą galima išspręsti pridėjus taimerį protokole nr. 2 kuris palauktų kažkokį laiko tarpą ir negavus patvirtinimo siųsti pakartotinai.
- Tačiau gali atsitikti taip, kad kadras bus išsiųstas du ar daugiau kartų.
- To išvengiama numeruojant kadrus. Kadrams numeruoti užtenka dviejų indeksų, pvz. 0 ir 1.

Vienkryptis protokolas Nr. 2 su taimeriu

1. Kompiuterio A tinklo sluoksnis perduoda paketą nr. 1 duomenų perdavimo kanalo sluoksniui. Paketas sėkmingai priimamas kompiuteryje B, išsiunčiamas patvirtinimas kompiuteriui A.
2. Patvirtinimo kadras yra pametamas perdavime dėl klaidų.
3. Kompiuteryje A, baigiasi taimerio laikas, negavus paketo nr. 1 kompiuteris A vėl siunčia paketą nr. 1 A kompiuteriui B.
4. Paketas nr. 1 vėl gaunamas kompiuteryje B ir duplikatas perduodamas tinklo sluoksniui kaip naujas paketas. Jeigu siunčiamas failas, duomenys jame bus sugadinti, nes tas pats duomenų blokas bus pakartotas du kartus.

```
/* 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 */
      }
 }
}
```

Figure 3-14. A positive acknowledgement with retransmission protocol.

Slenkančio lango protokolai

- Paprastai norima siųsti duomenis abiem kryptimis. Tokiu atveju ir patvirtinimai eitų abiem kryptimis.
- Galima panaudoti kanalą siunčiantį duomenų kadrus iš A į B perduoti patvirtinimams iš A į B.
- Norint sutaupyti duomenų apimties galima naudoti duomenų kadrus kartu ir patvirtinimamas naudojant specialų lauką duomenų kadre.
- Tokiu atveju reikia palaukti kol tikslo sluoksnis norės siųsti duomenis, kad į juos būtų galima įdėti patvirtinimą.

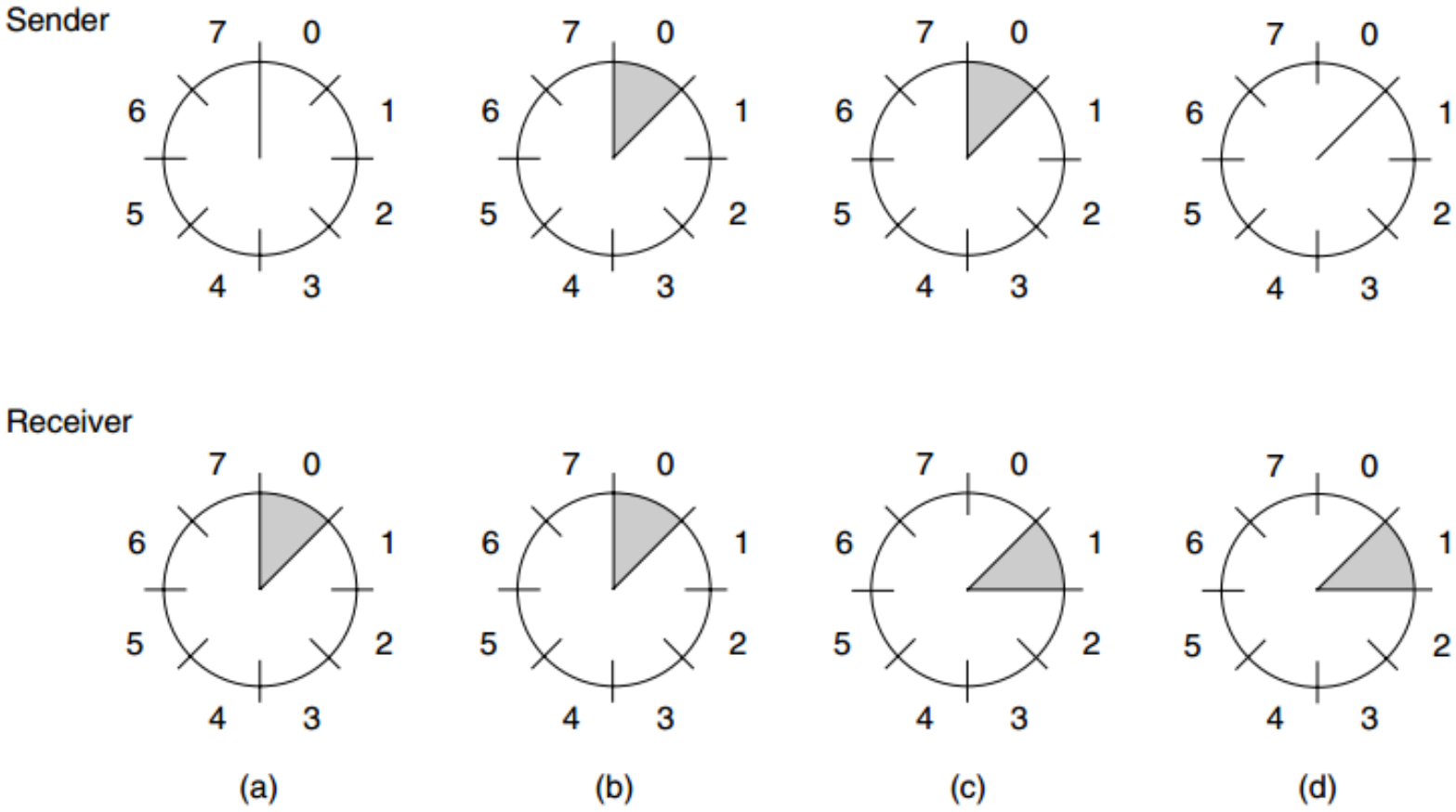


Figure 3-15. A sliding window of size 1, with a 3-bit sequence number. (a) Initially. (b) After the first frame has been sent. (c) After the first frame has been received. (d) After the first acknowledgement has been received.

Vieno bito slenkančio lango protokolas nr. 4

```
/* 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 */
}
}
```

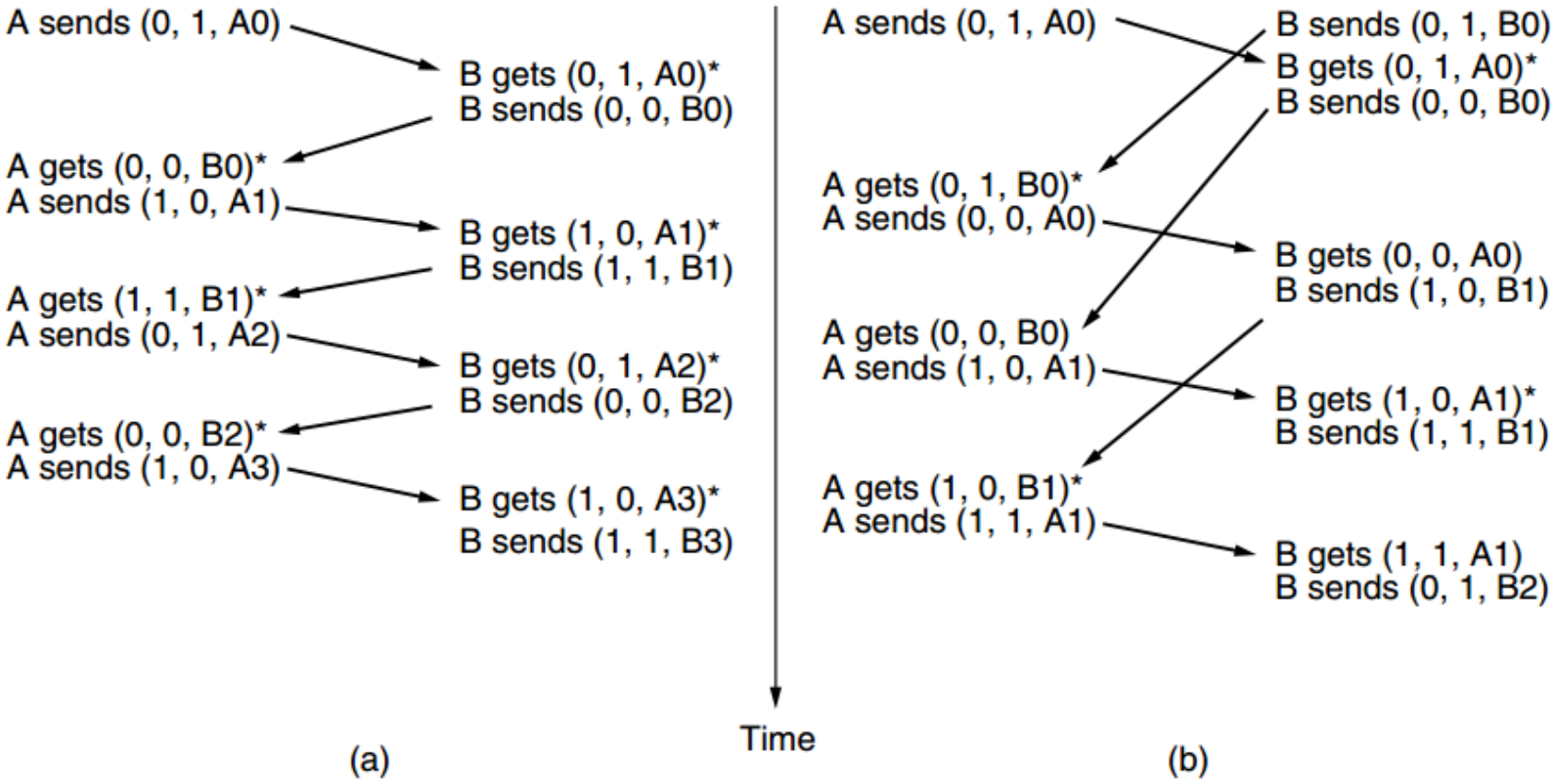


Figure 3-17. Two scenarios for protocol 4. (a) Normal case. (b) Abnormal case. The notation is (seq, ack, packet number). An asterisk indicates where a network layer accepts a packet.

Go-back-n protokolas nr. 5

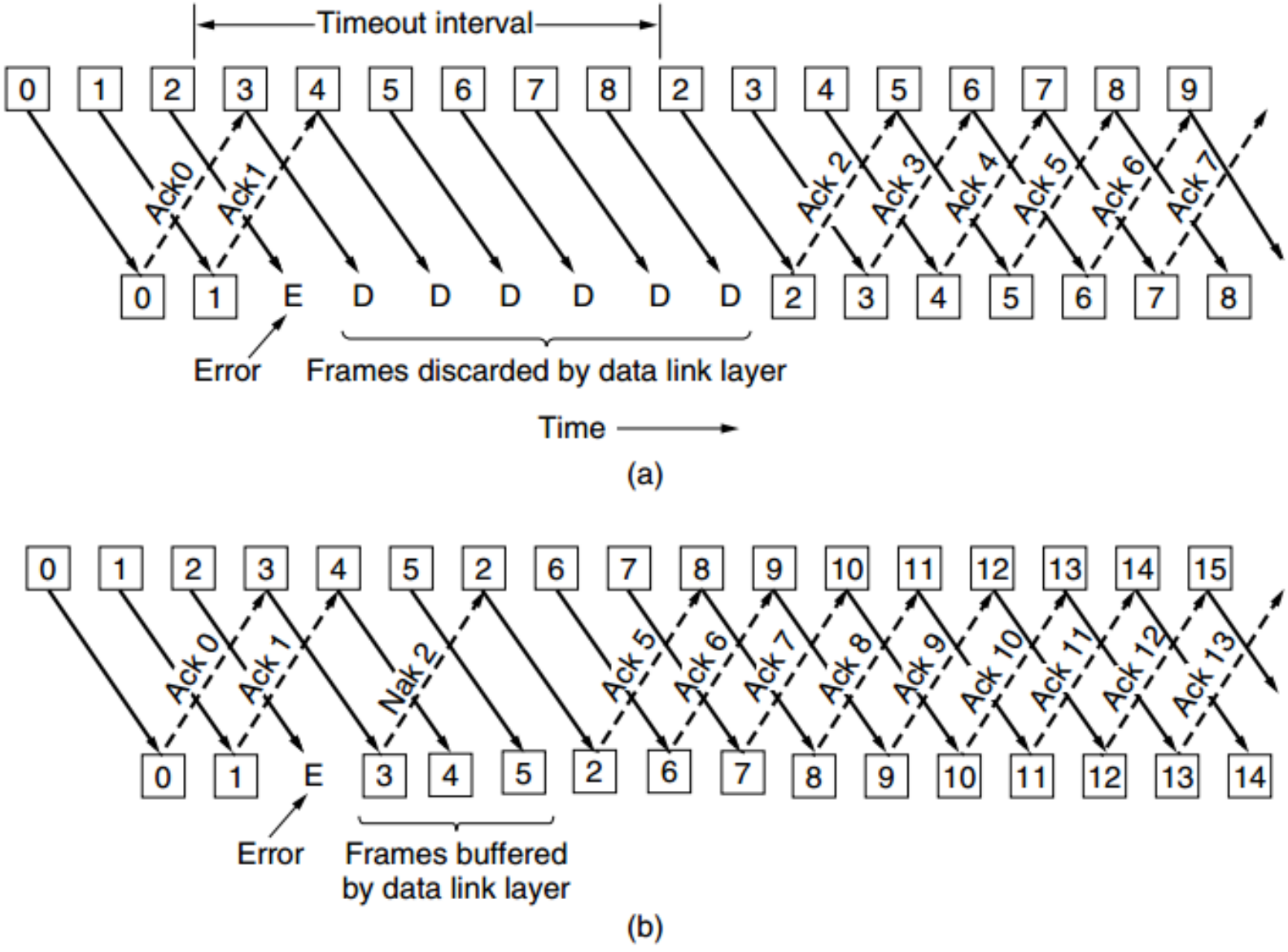


Figure 3-18. Pipelining and error recovery. Effect of an error when (a) receiver's window size is 1 and (b) receiver's window size is large.

```
/* 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
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;                   /* 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 */

  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 frame */
                        inc(next_frame_to_send);        /* prepare to send the next one */
                }

    }

    if (nbuffered < MAX_SEQ)
            enable_network_layer();
    else
            disable_network_layer();
  }
}
```

Figure 3-19. A sliding window protocol using go-back-n.

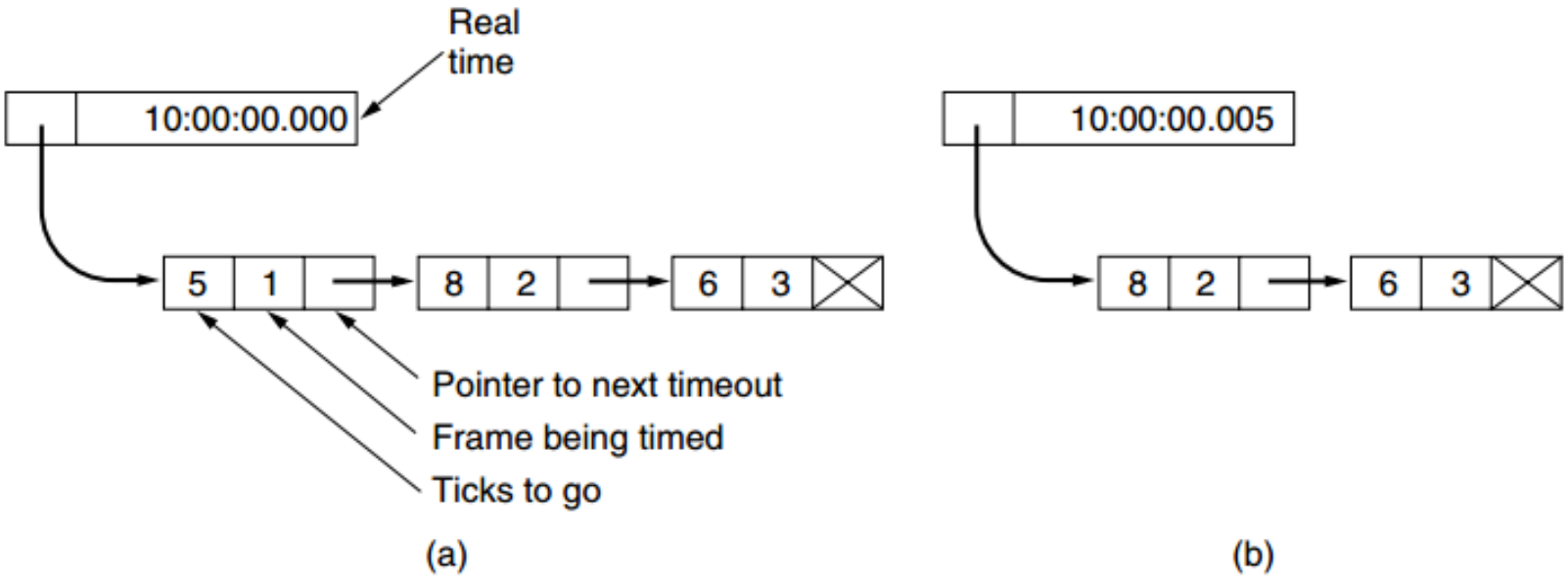


Figure 3-20. Simulation of multiple timers in software. (a) The queued timeouts. (b) The situation after the first timeout has expired.

Selective repeat protokolas nr. 6

```
/* 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 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 protocol 5, 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;
```

```
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();
  }
}
```

Figure 3-21. A sliding window protocol using selective repeat.

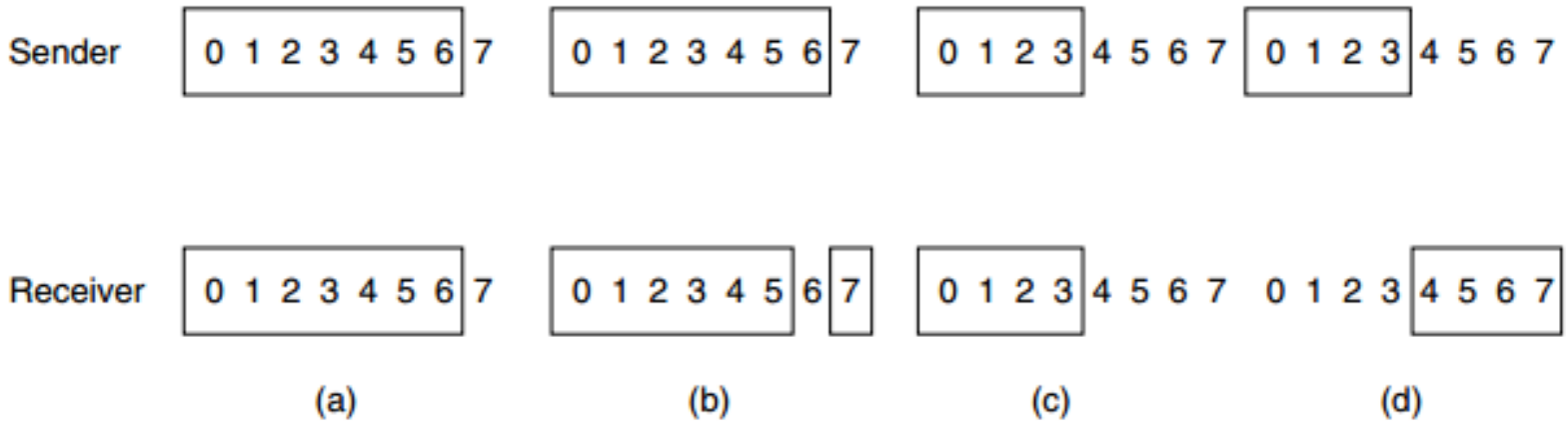


Figure 3-22. (a) Initial situation with a window of size7. (b) After 7 frames have been sent and received but not acknowledged. (c) Initial situation with a window size of 4. (d) After 4 frames have been sent and received but not acknowledged.

Uždaviniai

Uždaviniai (I)

21. Aukštesnio sluoksnio paketas padalinamas į 10 duomenų kadrų kurių kiekvienas nugabenamas nesugadintas su 80% tikimybe. Jeigu duomenų perdavimo kanalo sluoksnis neatlieka jokio klaidų taisymo ar aptikimo, kiek kartų vidutiniškai reikės siųsti paketą kol jis bus teisingai nusiųstas?
22. Norima perduoti simbolių seką A B ESC C ESC FLAG FLAG D, kaip atrodys duomenys su jais atlikus aptartą papildymo ESC baitais algoritmą.
23. Norima perduoti bitų seką 0111101111101111110, kaip ji atrodys po papildymo bitais pagal aptartą “bit stuffing” metodą.

Uždaviniai (II)

24. Sakėme kad reikia naudoti FLAG baitus žymėti duomenų kadro pabaigą ir pradžią. Taip darant, ties dviejų kadrų riba iš eilės eis du FLAG baitai. Ar užtektų naudoti vieną?
25. Tarkime žinutė 1001110010100011 yra perduodama naudojant Internet Checksum kontrolinę sumą (4 bitų). Kokia kontrolinės sumos reikšmė?
26. Kanalo pralaidumas yra 4 kbps, perdavimo uždelsimas yra 20 msec. Kokiam duomenų kadrų dydžių ruožui **stop-and-wait** protokolas išnaudos bent 50% kanalo pralaidumo?

Uždaviniai (III)

27. Ar įmanoma, kad protokole nr. 3 siuntėjas bandys įjungti taimerį jam jau veikiant. Jeigu taip, kaip tai gali įvykti? Jeigu ne, kodėl?
28. Jeigu protokole nr. 5 procedūroje “between” būtų naudojama sąlyga $a \leq b \leq c$ o ne $a \leq b < c$ kaip dabar, kokią tai įtaką turėtų protokolo efektyvumui ir korektiškumui? Paaiškinkite atsakymą.

Kitą kartą - Ethernet