КОМПЬЮТЕРНЫЕ СЕТИ

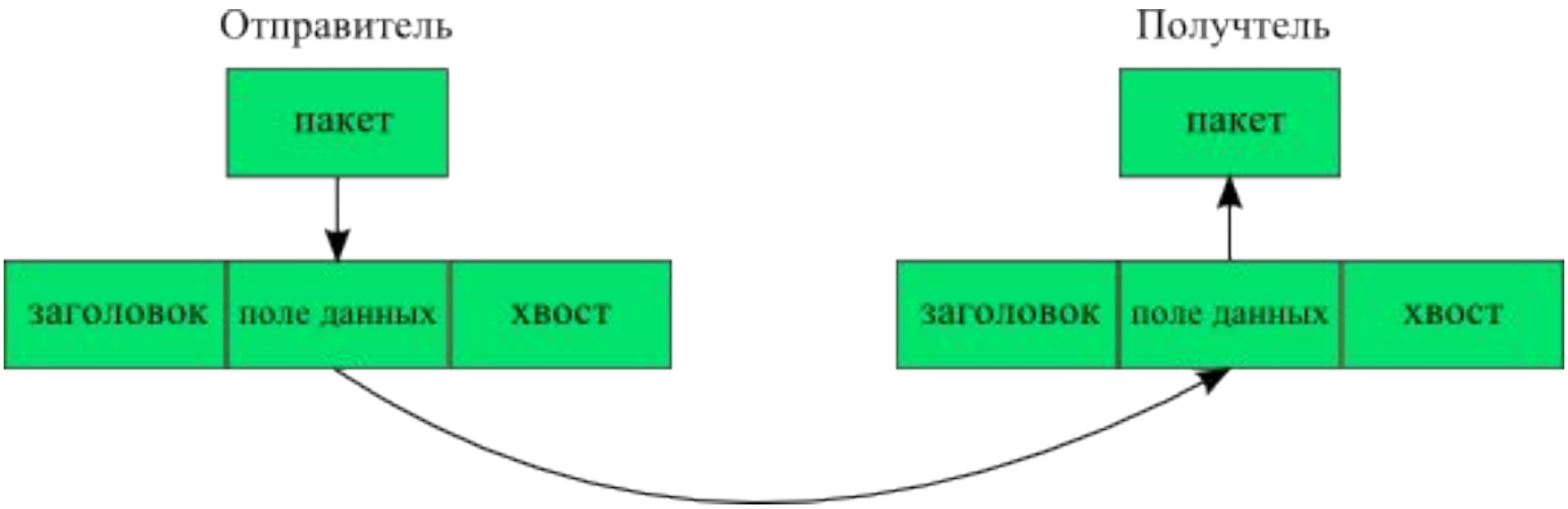
Лекция №5

Уровень передачи данных или канальный уровень.

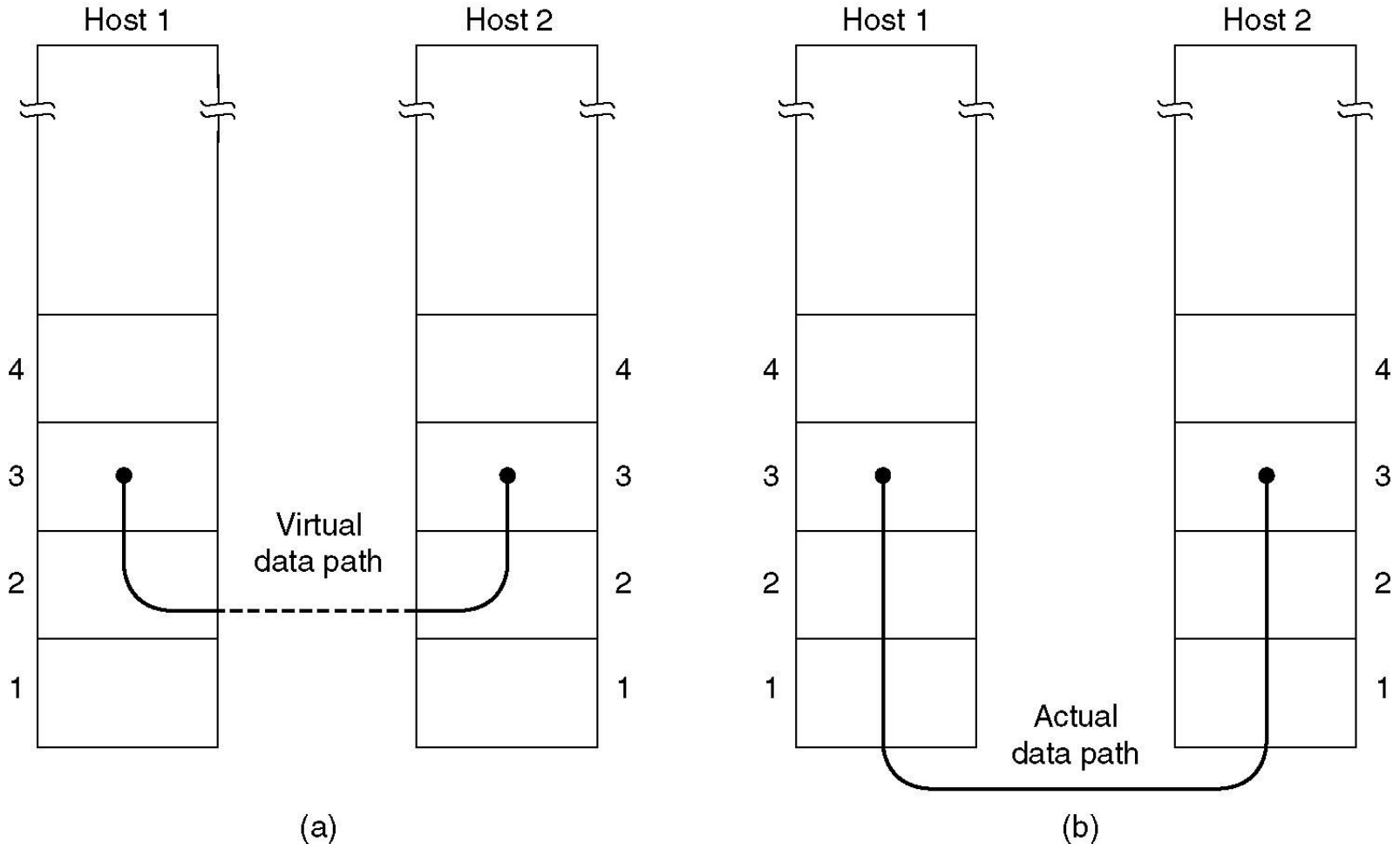
Уровень передачи данных

- Уровень передачи данных или канальный уровень (Data link layer)
- Функции:
  - обеспечение служебного интерфейса для сетевого уровня;
  - обработка ошибок передачи данных;
  - управление потоком данных.
- Сетевой уровень оперирует пакетами, канальный уровень формирует из пакетов **кадры** для передачи.
- Контроль ошибок, контроль потока может происходить и на более верхних уровнях.

Уровень передачи данных



Уровень передачи данных



Сервисы, предоставляемые сетевому уровню

- **Сервис без подтверждений, без установки соединения.**
  - Просто шлём независимые кадры, не обращая внимания на ошибки.
  - Применяется зачастую в локальных сетях.

- **Сервис с подтверждениями, без установки соединения.**
  - Подтверждение доставки кадра (квитанция).
  - Повторная посылка по тайм-ауту.
  - Можно переложить на сетевой уровень, но там пакеты больше

Сервисы, предоставляемые сетевому уровню

- **Сервис с подтверждениями, ориентированный на соединения.**

  - Первая фаза. Установка соединения, инициализация переменных и счетчиков для слежения за тем, какие кадры уже приняты, а какие ещё нет.
  - Вторая фаза. Передача кадров.
  - Третья фаза. Разрыв соединения, освобождение временных ресурсов.

Канальный уровень и физический уровень

- Физический уровень, предоставляющий интерфейс для канального, фактически принимает поток битов и пытается передать. Возможны ошибки при передаче. Количество принятых бит может отличаться от количества переданных.
- Задача канального уровня – обнаружить и исправить ошибки.
- Канальный уровень разбивает поток битов на отдельные кадры. Для каждого кадра подсчитывается контрольная сумма, которая проверяется после передачи. Ошибочный кадр игнорируется или данные о нем передаются отправляющей стороне.

Формирование кадра

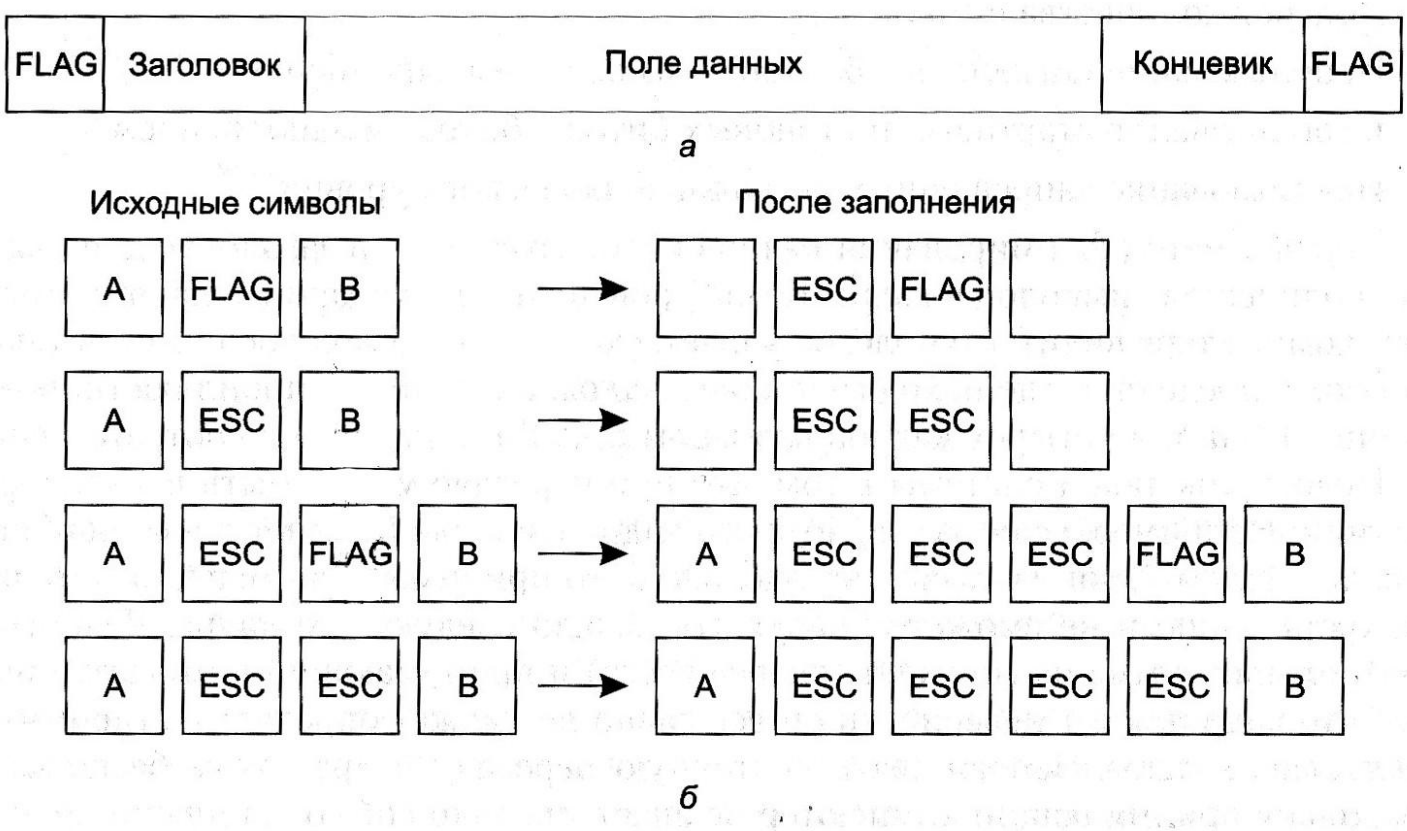
- Задача разбиения потока битов на отдельные кадры не решается простым временным интервалом.
- Методы маркировки начала и конца кадра:

1. Подсчёт количества символов.
2. Использование сигнальных байтов с символьным заполнением.
3. Использование стартовых и стоповых бит с битовым заполнением.
4. Использование запрещённых сигналов физического уровня.

Подсчёт количества символов

При искажении счетчика теряется синхронизация.

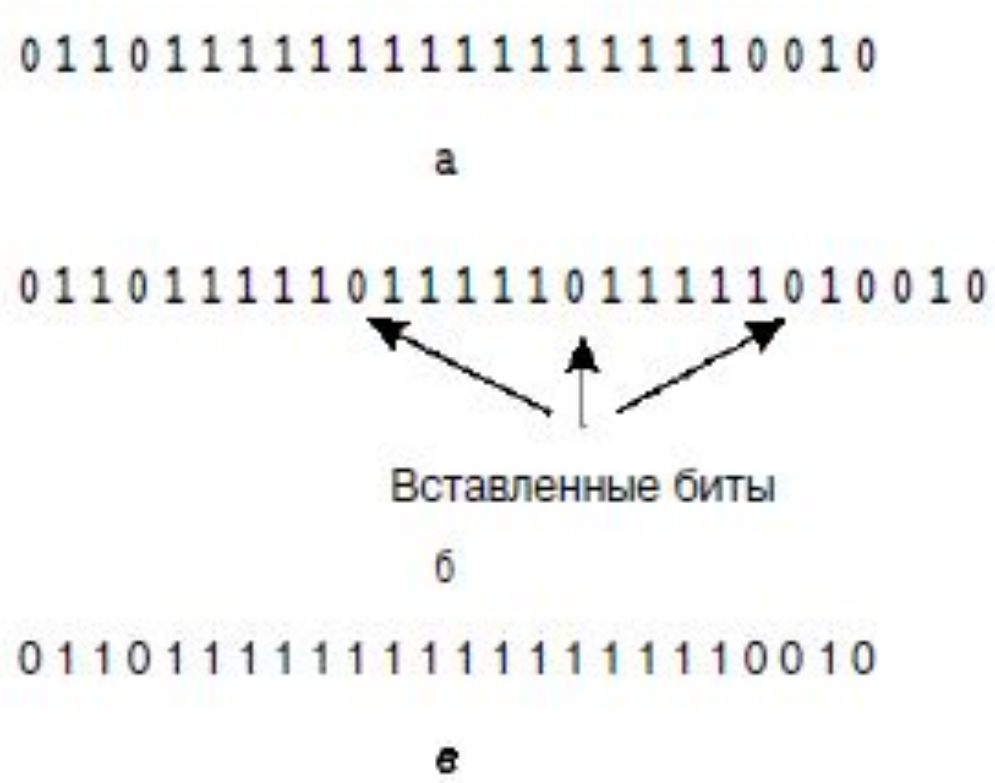
Использование сигнальных байтов с символьным заполнением



ESC – знак переключения кода. Уровень передачи данных убирает ESC перед передачей сетевому уровню.

Использование стартовых и стоповых битов с битовым заполнением

- Каждый кадр начинается и заканчивается последовательностью 01111110. Если при передаче в потоке 5 единиц – вставляется 0. Приемник восстанавливает, убирая лишние 0 в потоке.



Использование запрещённых сигналов физического уровня

- Если бит данных кодируется двумя физическими битами, то используется 2 комбинации уровней (чаще положительный и отрицательны переходы), а 2 оставшихся не используются. Их можно использовать в качестве границ кадров.
- Если используется избыточное кодирование, некоторые символы можно использовать как служебные, в том числе, и для ограничения кадра.

Обработка ошибок и управление потоком

- Необходимо гарантировать сетевому уровню доставку всех кадров и их порядок.
  - Используются служебные кадры для обратной связи с результатами доставки каждого кадра.
  - Для контроля полной потери используется таймер.
  - Необходима нумерация кадров для предотвращения повторной доставки.
- Управление потоком:
  - Управление потоком с обратной связью;
  - Управление потоком с ограничением (не применяется на канальном уровне).

Обнаружение и исправление ошибок. Корректирующее кодирование

- Две стратегии в компьютерных сетях:
  - Коды с исправлением ошибок или корректирующие коды;
  - Коды с обнаружением ошибок или коды с прямым исправлением ошибок.

- В некоторых сетях целесообразнее повторно запросить кадр, в других (особенно с большим количеством ошибок и малой скоростью) – восстановить исходную информацию, если это возможно.

Обнаружение и исправление ошибок. Корректирующее кодирование

- Пусть кадр состоит из m битов, и r избыточных (контрольных) битов. Тогда $n = m + r$ полная длина кадра. Набор из n битов называют **n-битовым кодовым словом или кодовой комбинацией.**

- **Кодовое расстояние** в понимании Хэмминга:

10001001

10110001

00111000

- **d = 3**

Обнаружение и исправление ошибок. Корректирующее кодирование

- В большинстве приложений передачи данных все 2^m возможных сообщений допустимы, но при добавлении контрольных битов не все 2^n возможных кодовых слов допустимы.
- Можно построить полный список всех кодовых слов.
- Минимальное кодовое расстояние в этом списке называется **минимальным кодовым расстоянием кода** (или расстоянием всего кода в смысле Хэмминга).

Обнаружение и исправление ошибок. Корректирующее кодирование

- Способности кода по обнаружению и исправлению ошибок зависят от его минимального кодового расстояния. Для обнаружения k ошибок в одном кодовом слове, необходим код с минимальным кодовым расстоянием $k+1$.
- Для исправления k ошибок в кодовом слове требуется код с минимальным расстоянием $2k+1$.
- Простейший пример кода с обнаружением ошибок – бит четности.

Обнаружение и исправление ошибок. Корректирующее кодирование

- Пусть требуется создать код, способный исправлять одиночные ошибки, состоящий из m информационных и r контрольных бит. Тогда справедливо неравенство:

$$(m+r+1) \leq 2^r$$

- При заданном m неравенство описывает минимальное требуемое количество r для возможности исправления одиночных ошибок.

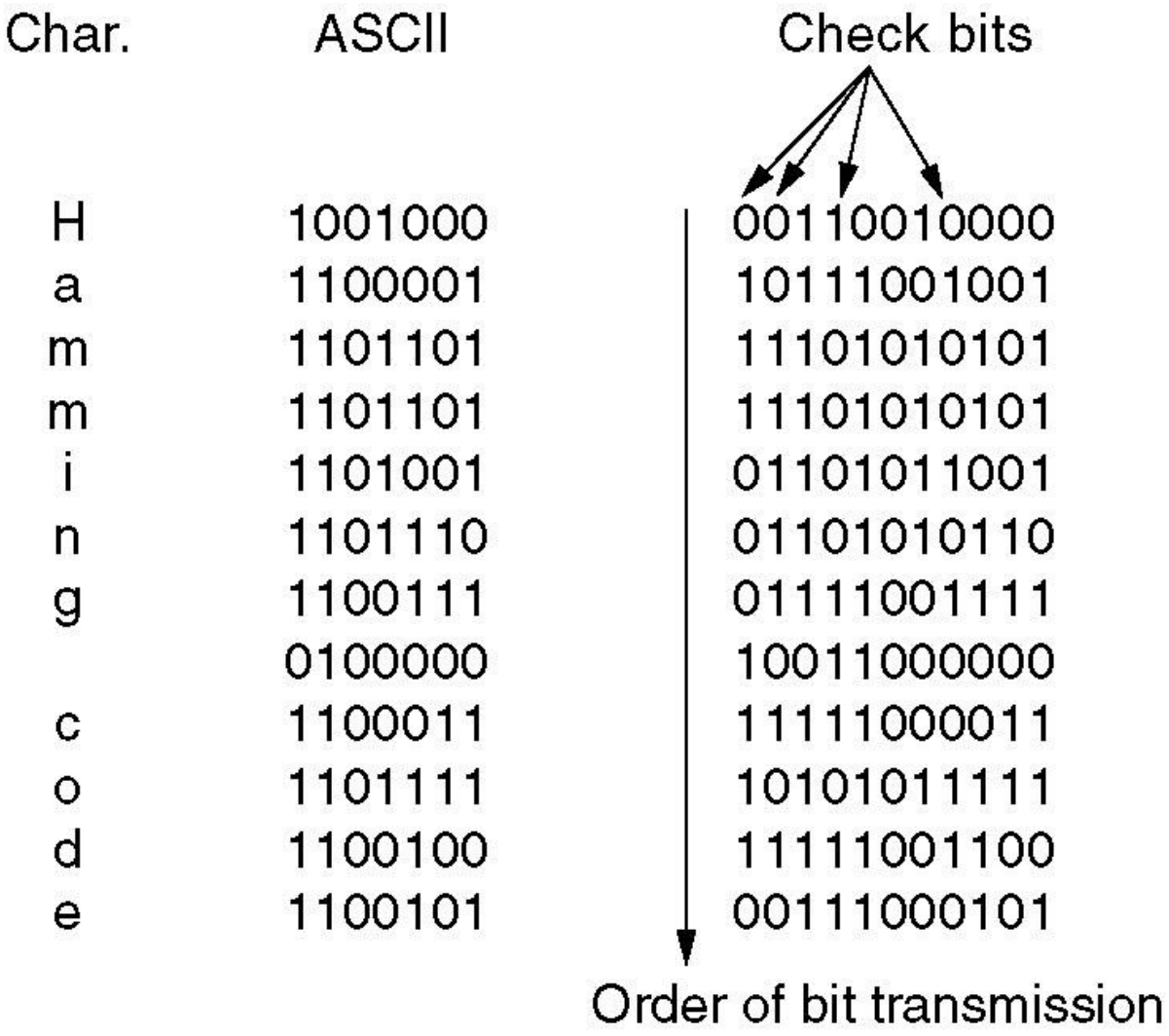
Обнаружение и исправление ошибок. Код Хэмминга

- Биты кодового слова нумеруются последовательно слева, начиная с 1.
- Биты с номерами, равными степени 2 (1,2,4,8,16,32,…), являются контрольными.
- Остальные биты заполняются данными.
- Каждый контрольный бит обеспечивает четность некоторой группы битов, включая себя.
- Каждый k-й бит будет проверяться контрольными битами, входящими в разложение k по степеням числа 2.

Обнаружение и исправление ошибок. Корректирующее кодирование

- Обнуляем счетчик, проверяем контрольные биты на четность. Если сумма k-го бита не четна – добавляем к счётчику. Если счётчик равен 0 – кодовое слово цело, иначе, в случае единичной ошибки, счетчик будет содержать номер инвертированного бита.
- Есть возможность целый блок ошибок. Выписываем k кодовых слов в виде матрицы, передаём по колонкам. Требуется kr проверочных бит. Блок из km бит может выдержать пакет ошибок длиной не более k бит.

Обнаружение и исправление ошибок. Корректирующее кодирование



Обнаружение и исправление ошибок. Коды с обнаружением ошибок

- Если к каждому блоку добавлять бит четности, вероятность обнаружения пакета ошибок, будет составлять всего 0,5. Можно увеличить вероятность, рассчитывая биты четности по столбцам матрицы, составленной из блока, который впоследствии передаётся по строкам. Тогда вероятность необнаружения составит 2^{-n} . Где n – число столбцов.
- На практике чаще всего используются CRC (Cyclic Redundancy Check – циклический избыточный код), или же полиномиальный код.

Элементарные протоколы передачи данных

- Неограниченный симплексный протокол
- Симплексный протокол с ожиданием
- Симплексный протокол для зашумленных каналов

Элементарные протоколы передачи данных. Общие объявления

```
#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
```

Элементарные протоколы передачи данных. Неограниченный симплексный протокол

- Данные передаются в одном направлении
- Сетевой уровень на передающей и принимающей стороне в состоянии постоянной готовности
- Время обработки равно 0
- Размер буфера не ограничен
- Канал связи между уровнями передачи данных идеален

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

Элементарные протоколы передачи данных. симплексный протокол с ожиданием

- Пусть сетевой уровень не может моментально обрабатывать данные (получающий уровень передачи данных не имеет неограниченного буферного пространства).
- Возможна ситуация, когда отправитель посылает быстрее, чем получатель обрабатывает.
- Решение – или жесткое ограничение или обратная связь служебным кадром, разрешающим дальнейшую отправку.

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

Элементарные протоколы передачи данных. симплексный протокол для зашумленных каналов

- Канал не идеален, кадры могут искажаться и теряться.
- Теряться и искажаться могут в том числе и служебные кадры.
- Требуется ввести порядковые номера и вести учёт номеров как на передающей, так и на принимающей стороне.

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

Элементарные протоколы передачи данных. Протоколы скользящего окна

- Дуплексная передача по одному каналу в обоих направлениях.
- Служебные и кадры данных можно различать по полю kind.
- Подтверждение можно добавлять к следующему на отправку от сетевого уровня пакету (ack в заголовке кадра). Piggybacking. Требуется таймер.

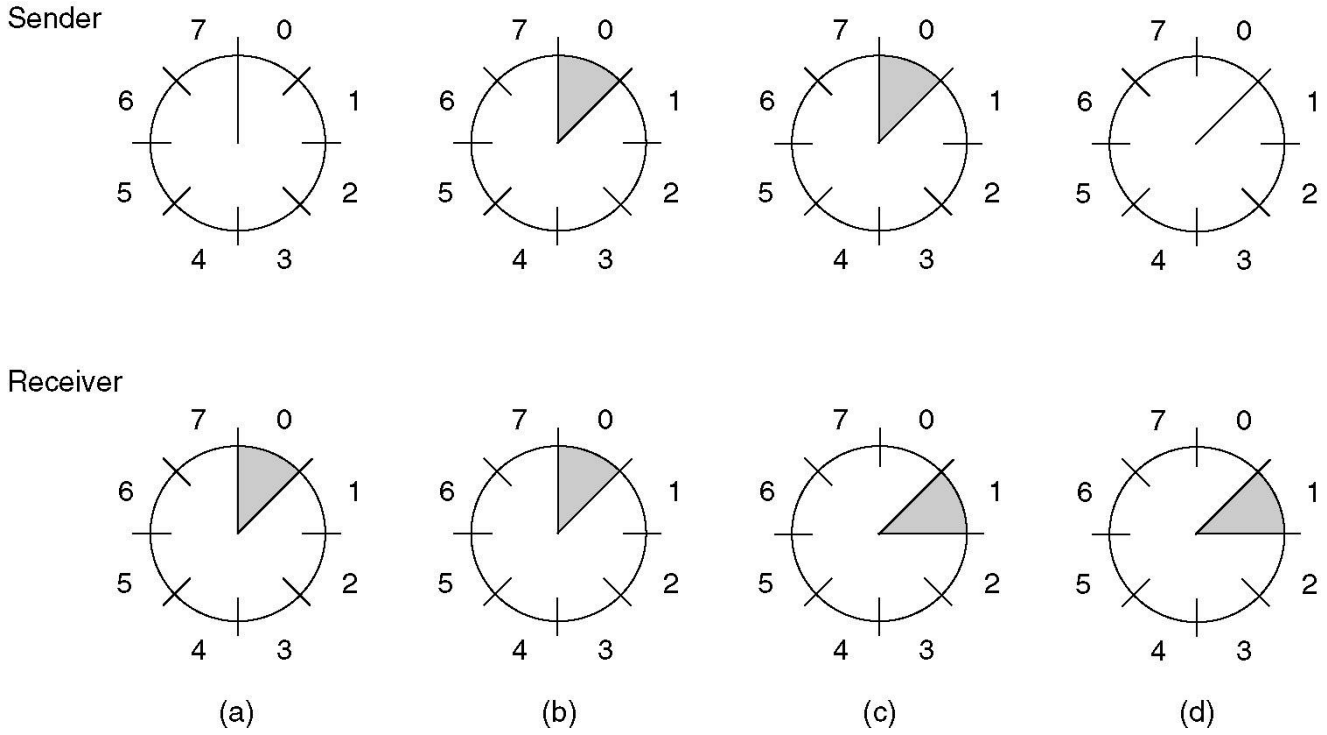
Элементарные протоколы передачи данных. Протоколы скользящего окна

- В любой момент времени отправитель работает с определённым набором порядковых номеров, соответствующих кадрам, которые разрешено посылать. **Посылающее окно.**
- Аналогично получатель работает с **принимающим окном.**
- Окно получателя и отправителя могут иметь разные границы и размеры.
- При поступлении от сетевого уровня пакета на отправку ему дается наибольше номер, верхняя граница окна увеличивается
- При получении подтверждения, нижняя граница уменьшается
- В окне отправителя – кадры, которые отправлены, но нет пока подтверждения.

Элементарные протоколы передачи данных. Протоколы скользящего окна

- Окно принимающего соответствует кадрам, которые он может принять. Кадр, не попадающий в окно – игнорируется.
- Прибывающий кадр с порядковым номером нижней границы предаётся на сетевой уровень. Формируется подтверждение, окно сдвигается на позицию.
- Окно единичного размера означает, что уровень передачи принимает кадры только по порядку.

Элементарные протоколы передачи данных. Протоколы скользящего окна



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.

Протокол однобитового скользящего окна

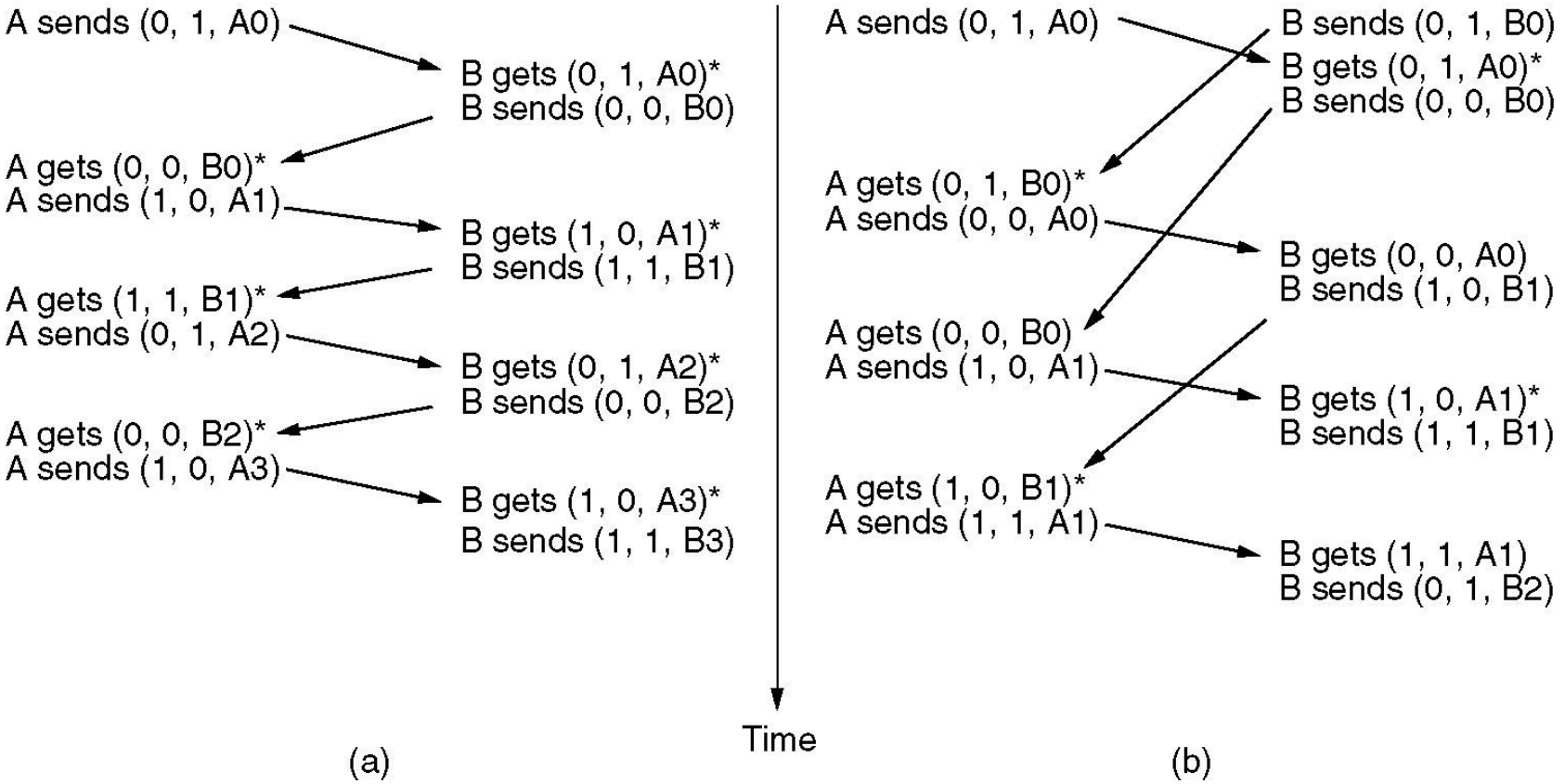
- Окно принимающего соответствует кадрам, которые он может принять. Кадр, не попадающий в окно – игнорируется.
- Прибывающий кадр с порядковым номером нижней границы предаётся на сетевой уровень. Формируется подтверждение, окно сдвигается на позицию.
- Окно единичного размера означает, что уровень передачи принимает кадры только по порядку.

Протокол однобитового скользящего окна

- Метод ожидания, т.к. после посылки кадра отправитель ждет подтверждения.
- next_frame_to_send номер кадра, который отправитель пытается послать.
- frame_expected номер кадра, ожидаемого получателем.
- Только один уровень передачи данных может начинать передачу.

```
/* 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 */
```

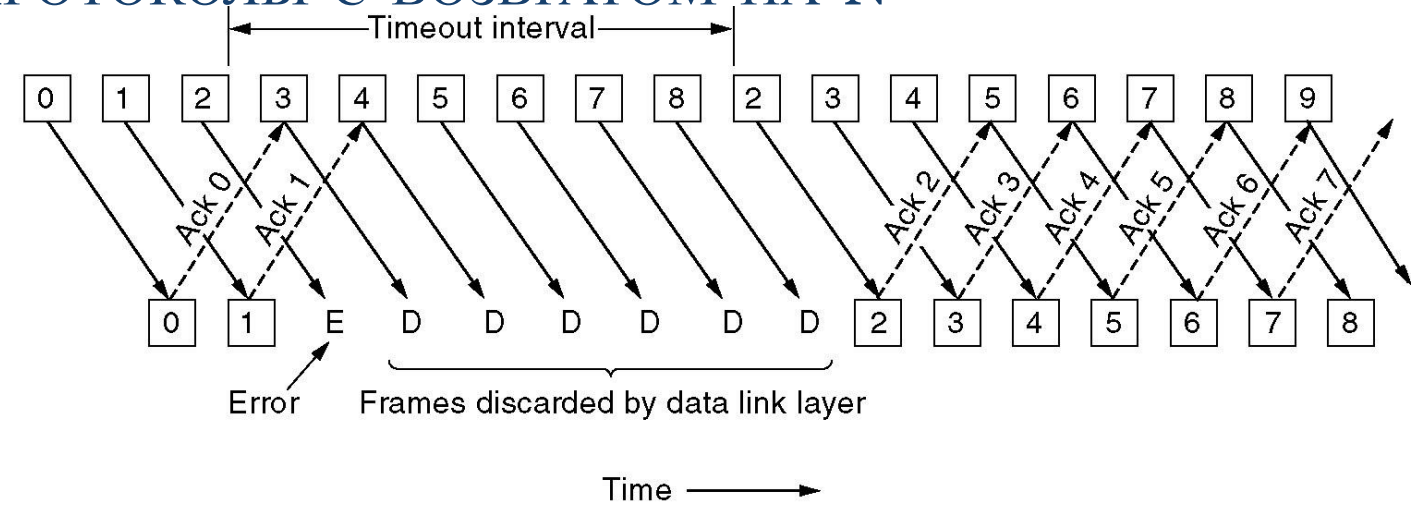
```
while (true) {
    wait_for_event(&event);                 /* frame_arrival, cksum_err, or timeout */
    if (event == frame_arrival) {           /* a frame has arrived undamaged. */
        from_physical_layer(&r);            /* go get it */

        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 senderís 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 sends (0, 1, A0)
B gets (0, 1, A0)*
B sends (0, 0, B0)
A gets (0, 0, B0)*
A sends (1, 0, A1)
B gets (1, 0, A1)*
B sends (1, 1, B1)
A gets (1, 1, B1)*
A sends (0, 1, A2)
B gets (0, 1, A2)*
B sends (0, 0, B2)
A gets (0, 0, B2)*
A sends (1, 0, A3)
B gets (1, 0, A3)*
B sends (1, 1, B3)
Time
(a)
A sends (0, 1, A0)
B sends (0, 1, B0)
B gets (0, 1, A0)*
B sends (0, 0, B0)
A gets (0, 1, B0)*
A sends (0, 0, A0)
B gets (0, 0, A0)
B sends (1, 0, B1)
A gets (0, 0, B0)
A sends (1, 0, A1)
B gets (1, 0, A1)*
B sends (1, 1, B1)
A gets (1, 0, B1)*
A sends (1, 1, A1)
B gets (1, 1, A1)
B sends (0, 1, B2)
(b)

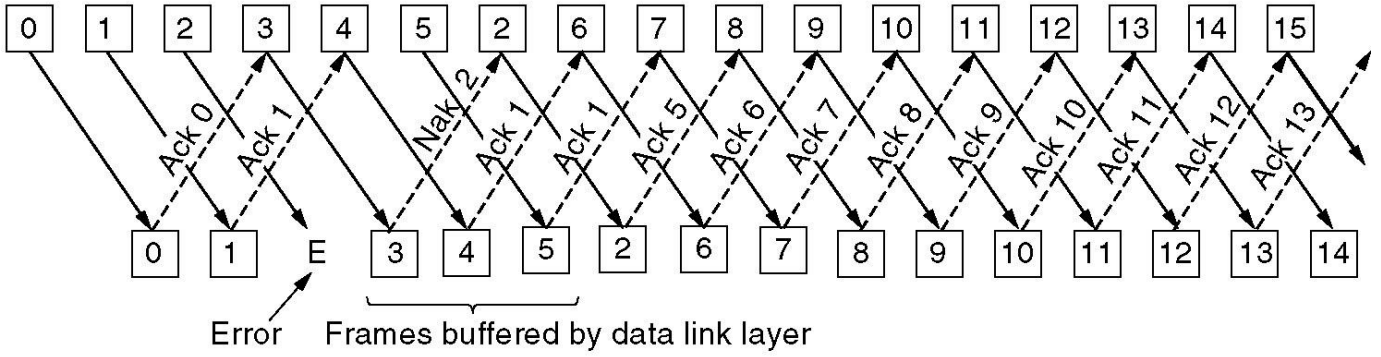
Протоколы с возвратом на N

- Ранее считалось, что время на передачу кадра от отправителя к получателю и на обратную отправку подтверждения было пренебрежимо мало. В случае, если это не так, эффективность очень мала.
- Разрешим отправителю слать не 1 кадра а w. Конвейерная обработка.
- Возможна ошибка при передаче кадра внутри этого w:
  - Возврат на n – игнорируем все кадры с номером больше поврежденного.
  - Выборочный повтор – буферизуем нормальные, как получаем повторно битый – высылаем максимальный ACK. Есть модификации с пересылкой NAK, стимулирующего повторную отправку.

Протоколы с возвратом на N



(a)



(b)

```
/* Protocol 5 (pipelining) allows multiple outstanding frames. The sender may transmit up
   to MAX_SEQ frames without waiting for an ack. In addition, unlike the previous protocols,
   the network layer is not assumed to have a new packet all the time. Instead, the
   network layer causes a network_layer_ready event when there is a packet to send. */

#define MAX_SEQ 7                       /* should be 2^n – 1 */
typedef enum {frame_arrival, cksum_err, timeout, network_layer_ready} event_type;
#include "protocol.h"

static boolean between(seq_nr a, seq_nr b, seq_nr c)
{
/* Return true if a <=b < c circularly; false otherwise. */
  if (((a <= b) && (b < c)) || ((c < a) && (a <= b)) || ((b < c) && (c < a)))
        return(true);
    else
        return(false);
}

static void send_data(seq_nr frame_nr, seq_nr frame_expected, packet buffer[ ])
{
/* Construct and send a data frame. */
  frame s;                              /* scratch variable */

  s.info = buffer[frame_nr];            /* insert packet into frame */
  s.seq = frame_nr;                     /* insert sequence number into frame */
  s.ack = (frame_expected + 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 */
```

```
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();
   }
}
```

Протоколы с выборочным повтором

- Позволяет более эффективно использовать ресурсы при частых ошибках.

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

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

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

Протоколы с выборочным повтором

- Возможно перекрытие сдвинутым окном исходного.
- Размер окна должен не превышать половины от количества порядковых номеров.
- Количество буферов у получателя равно размеру окна.

Верификация протоколов. Конечные автоматы

- Протокольная машина – отправитель и получатель. Состояние протокольной машины – все значения всех её переменных, в том числе и программные счётчики.
- Состояние канала определяется его содержимым.
- Каждое состояние может быть соединено с другими переходами.
- Имея полное описание протокольной машины можно получить двунаправленный граф.
- Методы теории графов (транзитивное замыкание) позволяют проводить анализ достижимости.

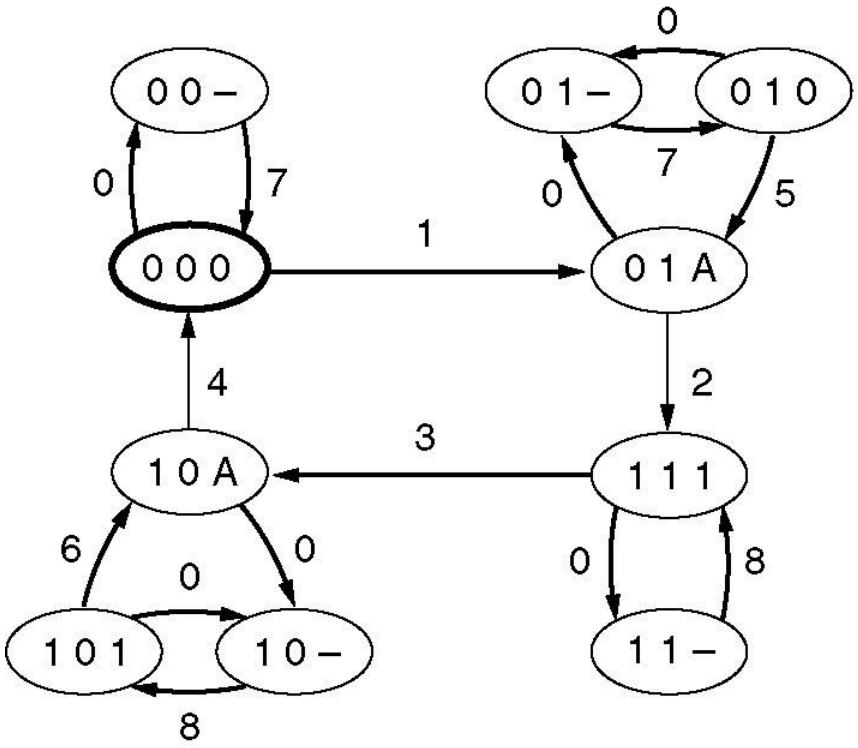
Верификация протоколов. Конечные автоматы

- (S,M,I,T)
- S – множество состояний, в которых могут находиться процессы и канал.
- M – множество кадров, передающихся по каналу.
- I – множество начальных состояний процессов.
- T – множество переходов между состояниями.

Верификация протоколов. Конечные автоматы

- Протокол с положительным подтверждением и повторной передачей.
- (S,R,C)
- S – 0 или 1, номер кадра, посылаемого отправителем.
- R – 0 или 1, номер кадра, ожидаемого получателем.
- C – состояние канала. 0 1 A -
- T – множество переходов между состояниями.

Верификация протоколов. Конечные автоматы



(a)

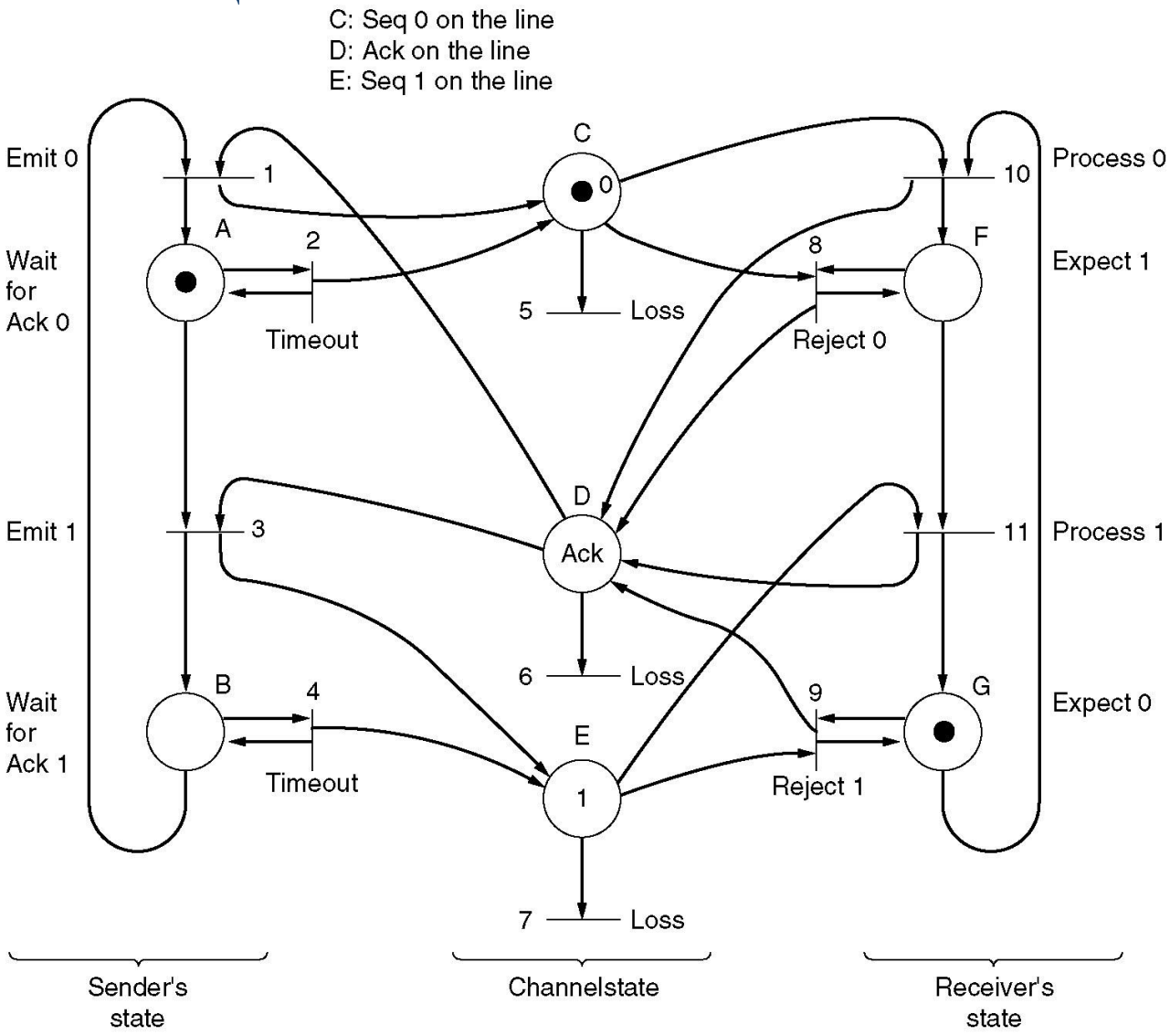
Transition	Who runs?	Frame accepted	Frame emitted	To network layer
0	–	(frame lost)		–
1	R	0	A	Yes
2	S	A	1	–
3	R	1	A	Yes
4	S	A	0	–
5	R	0	A	No
6	R	1	A	No
7	S	(timeout)	0	–
8	S	(timeout)	1	–

(b)

Верификация протоколов. Сети Петри

- Четыре элемента:
  - Позиция – состояние, в котором может находиться система или её часть.
  - Маркеры, помещающие состояние системы.
  - Переходы (каждый переход может иметь входящие и исходящие дуги)
- Переход называется разрешенным, если имеется маркер на входящей позиции.
- В отличии от автоматов, не составных состояний.
- Могут быть представлены в виде грамматики.

Верификация протоколов. Сети Петри



Верификация протоколов. Сети Петри

- Грамматика сети Петри:

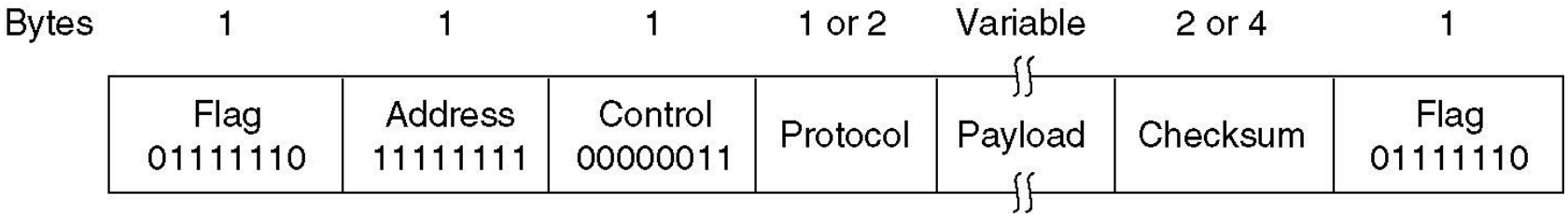
1. BD > AC
2. A > A
3. AD > BE
4. B > B
5. C >
6. D >
7. E >
8. CF > DF
9. EG > DG
10. CG > DF
11. EF > DG

Протокол PPP

- Point-to-Point Protocol RFC 1661 1662 1663
  - Позволяет обнаружение ошибок;
  - Поддерживает несколько протоколов;
  - Байт-ориентирован;
- Методы PPP:

1. Метод формирования кадров
2. Протокол управления каналом (установка канала, тестирование, обмен параметрами). Поддерживает синхронные и асинхронные линии, байт и бит ориентированное кодирование
3. Выработка параметров для сетевого уровня NCP Network Control Protocol

Протокол PPP



Протокол PPP