Chapter 3 The Data Link Layer

2019 Edition

Southeast University, Nanjing

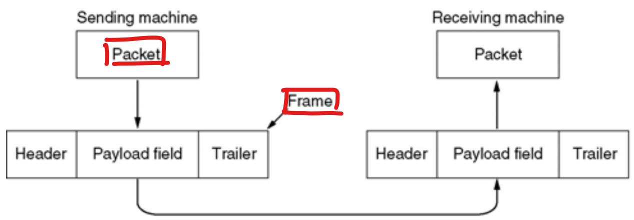
Data Link Layer Design Issues

- Services Provided to the Network Layer
- Framing
- Error Control
- Flow Control

Functions of the Data Link Layer

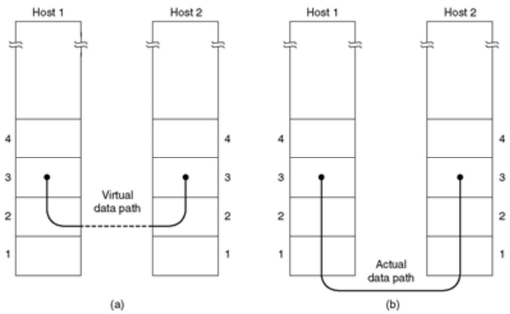
1 • **Provide service interface to the network layer**
2 • **Dealing with transmission errors**
3 • **Regulating data flow**
  – Slow receivers not swamped by fast senders

Functions of the Data Link Layer (2)



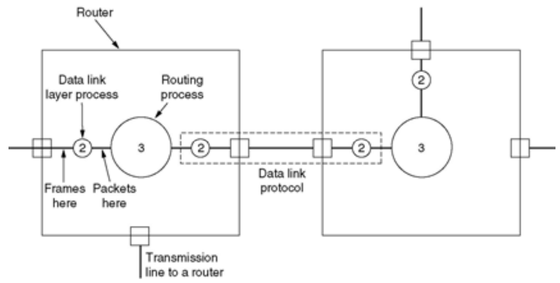
Relationship between packets and frames.

Services Provided to Network Layer



- (a) Virtual communication.
- (b) Actual communication.

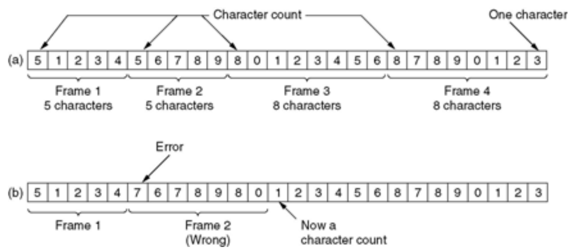
Services Provided to Network Layer (2)



Placement of the data link protocol.

Framing

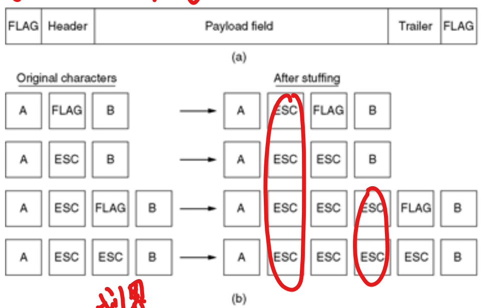
(1) 字符计数法



A character stream. (a) Without errors. (b) With one error.

Framing (2)

(2) byte stuffing



划界

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

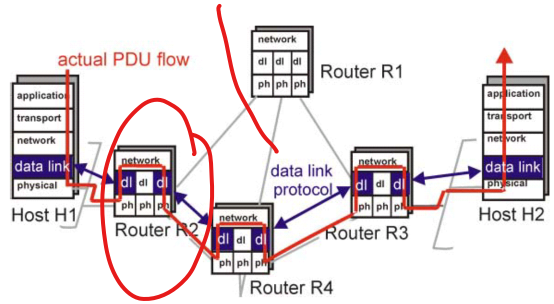
Framing (3)

(3) bit stuffing

```
(a) 0 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 1 0

(b) 0 1 1 0 1 1 1 1 1 0 1 1 1 1 1 0 1 1 1 1 1 0 1 0 0 1 0
                      ↑          ↑          ↑
                              Stuffed bits

(c) 0 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 1 0
```

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

Link Layer Protocols



Link Layer Services

- ① *Framing and link access*:
  - encapsulate datagram into frame adding header and trailer,
  - implement channel access if shared medium,
  - 'physical addresses' are used in frame headers to identify source and destination of frames on broadcast links
- ② *Reliable Delivery*:
  - seldom used on fiber optic, co-axial cable and some twisted pairs too due to low bit error rate.
  - Used on wireless links, where the goal is to reduce errors thus avoiding end-to-end retransmissions

Link Layer Services (more)

- ③ *Flow Control*:
  - pacing between senders and receivers
- ④ *Error Detection*:
  - errors are caused by signal attenuation and noise.
  - Receiver detects presence of errors:
  - it signals the sender for retransmission or just drops the corrupted frame
- ⑤ *Error Correction*:
  - mechanism for the receiver to locate and correct the error without resorting to retransmission

Link Layer Services (more)

Three type of service

- Acknowledged connect-oriented service
- Unacknowledged connectionless service
- Acknowledged connectionless service

Link Layer Protocol Implementation

- Link layer protocol entirely implemented in the adapter (eg,PCMCIA card). Adapter typically includes: RAM, DSP chips, host bus interface, and link interface
- Adapter **send** operations: encapsulates (set sequence numbers, feedback info, etc.), adds error detection bits, implements channel access for shared medium, transmits on link
- Adapter **receive** operations: error checking and correction, interrupts host to send frame up the protocol stack, updates state info regarding feedback to sender, sequence numbers, etc.

Framing

Framing is one of the basic function that Data link layer should do.

- Byte counting
- Byte stuffing
- Bit stuffing
- Usage of physical layer illegal encoding
- Etc…

Error Detection and Correction

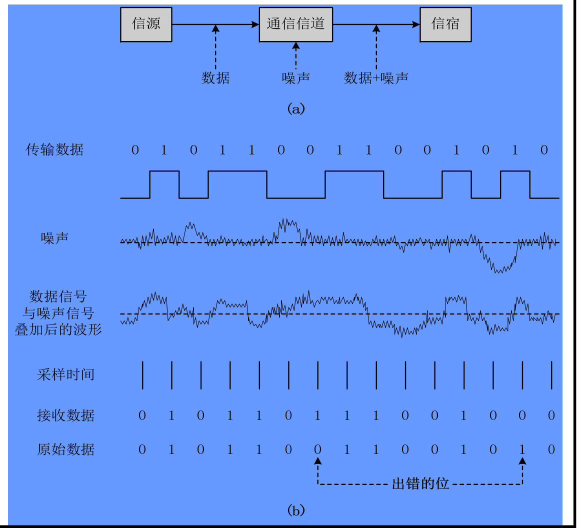
- Error-Correcting Codes
- Error-Detecting Codes

Errors

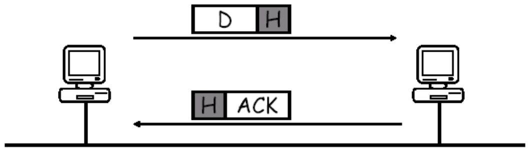
- A bit error occurs when a source sends a bit, b, and the destination receives NOT b.

  $i.e.\ \ b \rightarrow b \oplus 1.$
- The error can take place on the link (e.g. EM interference, or signal loss), or in the source or destination (e.g. failed hardware, or bit errors in memories).
- The bit error rate (BER) tells us the probability of any given bit being in error. Typical values are BER = 10^{-9} for an electrical link, and 10^{-12} for an optical link.

<u>Error Appearance</u>



Error Detection



An example:

Assume BER and independent errors,

Packet Error Rate = PER = 1 - (1 - BER)^N

PER ~= N (BER) if N (BER) << 1

e.g. N = 10^4, BER = 10^{-7} , PER = 10^{-3}

In practice, bit errors occur in BURSTS

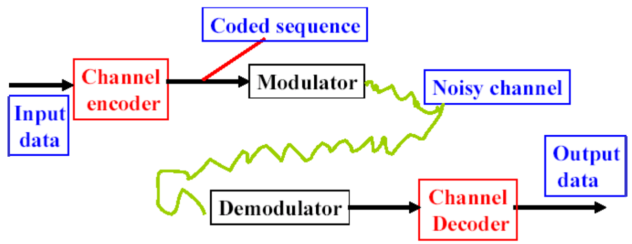
Error detection

Basic idea is to add redundant information

1. We use codes to help us detect errors.
2. The set of possible messages is mapped by a function onto the set of codes.
3. We pick the mapping function so that it is easy to detect errors among the resulting codes. (channel coding)
4. Naive algorithm 1- transmit each packet twice
   - But what happens if there is a difference? Which one is correct?
5. Naive algorithm 2- transmit each packet three times, take best 2 out of 3
   - Now I'm sending 200% overhead!
   - Still, errors can get through (how?)

- Error detection is not 100%;
- protocol may miss some errors, but rarely
- Larger EDC field yields better detection and correction

System with channel coding



Coding definitions

Coding is a function that takes *k* bit codewords and maps them (uniquely) to *n* bit codewords

- Code is length *n* and dimension *k*
- The rate of the code is k/n
- Set of all possible encoded messages is an error correcting code
- If errors change one valid codeword into another, we have a problem

Parity check code

Start with k bits, add another

- The rate of the code is k/(k+1)
- The code: make sure the number of ones in the message is even/odd(for even/odd parity check)
- Example:
- 011011101 ← Parity check bit

Properties:

and only detect

- Can detect an odd number of bit errors
- ~~Cannot correct any errors~~

Two dimensional parity

- Example: add 14 bits to 42 bit message

0101001	1
1101001	0
1011110	1
0001110	1
0110100	1
1011111	0
1111011	0

- Perform across rows and columns
- Catches all 1-, 2-, 3-, and some 4-bit errors
- Can correct all 1-bit errors

Hamming Distance

Number of bits that differ between two codes

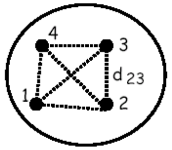
e.g. 1 0 0 1 0 1 0 1
1 0 1 1 1 0 0 1

0 0 1 0 1 1 0 0 → HD=3

In our example code (replicated bits), all codes have at least two bits different from every other code. Therefore, it has a Hamming distance of 2.

Hamming Distance

Set of codes



$HD = \min_{ij}(d_{ij})$

To reliably detect a d-bit error: HD > d $HD \geq d+1$
To reliably correct a d-bit error: HD > 2d

Error-Correcting Codes

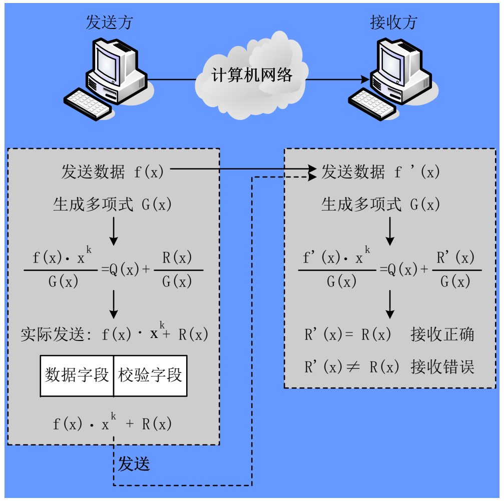
Char.	ASCII	Check bits
H	1001000	00110010000
a	1100001	10111001001
m	1101101	11101010101
m	1101101	11101010101
i	1101001	01101011001
n	1101110	01101010110
g	1100111	01111001111
	0100000	10011000000
c	1100011	11111000011
o	1101111	10101011111
d	1100100	11111001100
e	1100101	00111000101

Order of bit transmission

Use of a Hamming code to correct burst errors.

Checksums

- Basic idea: Add up the data and send the data and the sum
- Commonly used in the Internet (IP, ICMP, TCP, UDP)
- Typically used for detection only
- Not hard to implement in software



CRC

Cyclic Redundancy Check

- A much stronger algorithm
- Invented in 1962 (Peterson)
- Used in many protocols (Ethernet, ATM are two prominent examples)
- Typically implemented in hardware (XORs and shift registers)

- Notation: Think of (n+1) bit message as a n^{th} degree polynomial whose highest order term is x^n
- Example: $110101 = x^5 + x^4 + x^2 + 1$

Cyclic Redundancy Check

- Sender and receiver agree on divisor polynomial C(x) of degree k
  Example: $C(x) = x^3 + x^2 + 1$ -> k=3
- Algorithm: Given n bits of data, generate a k bit check sequence that gives a combined n+k bits that are divisible by a chosen divisor C(x)
  i.e., select the k bits such that the remainder is zero

Operation and Calculation

- The sender carries out on-line, in hardware the division of the string D by the polynomial G and appends the remainder R to it
- The receiver divides < D,R> by G; if the remainder is non-zero, the transmission was corrupted
- International standards for G polynomials of degrees 8, 12, 15 and 32 have been defined
- Message M(x)
- $T(x) = M(x) * x^k$ (add k zeros to end)
- Divisor C(x) of degree k
- Remainder R(x) of length k
- Use polynomial arithmetic, modulo 2

Calculating the CRC

```
         11111001
1101)10011010000
     1101
      1001
      1101
       1000
       1101
        1011
        1101
         1100
         1101
           1000
           1101
            101
```

Zero-pad to the degree of the CRC

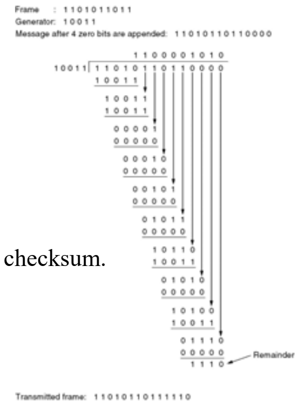
Here is my remainder

Therefore, send P(x):

$\underbrace{10011010}_{M(x)}\underbrace{101}_{R(x)}$

Error-Detecting Codes

Calculation of the polynomial code checksum.



Detect or Correct?

Advantages of Error Detection
- Requires smaller number of bits/overhead.
- Requires less/simpler processing.

Advantages of Error Correction
- Reduces number of retransmissions.

Most data networks today use error detection, not error correction.

Detect or Correct?

An example

Assume: 1. Packets are of lengths 923 bits
2. PER = 10^{-5}

Overhead of Error Correction:

Assume we use: BCH (1023, 923, 10)
Therefore, we send 923 data bits as 1023 bits.

Transmission Overhead = $\frac{100}{923}$ ~= 10%

Overhead of Error Detection:

Assume we use: 32-bit CRC; one retransmission per error.
Therefore, we send 923 data bits as 955 bits.

Transmission Overhead = $\frac{(923 + 32)\,10^{-5} + 32}{923}$ ~= 3%

Next section

Retransmission Protocols
--Simple LLC protocol (SWP)
--Sliding window protocols

Elementary Data Link Protocols

- An Unrestricted Simplex Protocol
- A Simplex Stop-and-Wait Protocol
- A Simplex Protocol for a Noisy Channel

Protocol Definitions

```
#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;
```

Some definitions needed in the protocols to follow.
These are located in the file protocol.h.

Protocol Definitions (ctd.)

```
/* 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
```

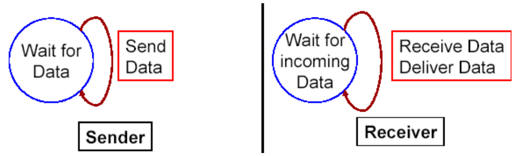
Some definitions needed in the protocols to follow. These are located in the file protocol.h.

Unrestricted Simplex Protocol

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

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

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

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

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

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

SLLC1.0

- Assume underlying channel perfectly reliable
  - no bit errors
  - no loss of packets
- Assume receiver always can pace up with the sender
  - no need for follow control
- separate FSMs for sender, receiver:
  - sender sends data into underlying channel
  - receiver read data from underlying channel



Simplex Stop-and-Wait Protocol

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

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

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

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

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

A Simplex Protocol for a Noisy Channel

A positive acknowledgement with retransmission protocol.

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

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

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

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

A Simplex Protocol for a Noisy Channel (ctd.)

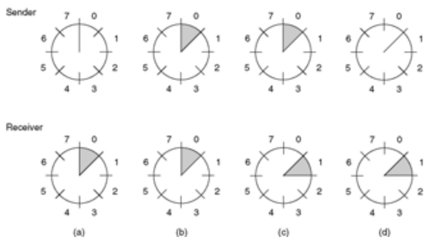
```
void receiver3(void)
{
 seq_nr frame_expected;
 frame r, s;
 event_type event;

 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 */
     }
 }
}
```

A positive acknowledgement with retransmission protocol.

Sliding Window Protocols

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

Sliding Window Protocols (2)



- 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.

A One-Bit Sliding Window Protocol

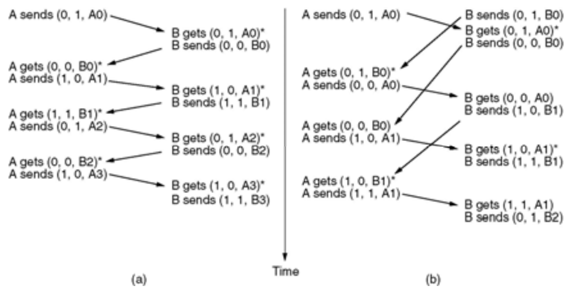
```
/* Protocol 4 (sliding window) is bidirectional. */
#define MAX_SEQ 1                        /* must be 1 for protocol 4 */
typedef enum {frame_arrival, cksum_err, timeout} event_type;
#include "protocol.h"
void protocol4 (void)
{
  seq_nr next_frame_to_send;             /* 0 or 1 only */
  seq_nr frame_expected;                 /* 0 or 1 only */
  frame r, s;                            /* scratch variables */
  packet buffer;                         /* current packet being sent */
  event_type event;

  next_frame_to_send = 0;                /* next frame on the outbound stream */
  frame_expected = 0;                    /* frame expected next */
  from_network_layer(&buffer);           /* fetch a packet from the network layer */
  s.info = buffer;                       /* prepare to send the initial frame */
  s.seq = next_frame_to_send;            /* insert sequence number into frame */
  s.ack = 1 – frame_expected;            /* piggybacked ack */
  to_physical_layer(&s);                 /* transmit the frame */
  start_timer(s.seq);                    /* start the timer running */
```

Continued →

A One-Bit Sliding Window Protocol (ctd.)

```
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 One-Bit Sliding Window Protocol (2)

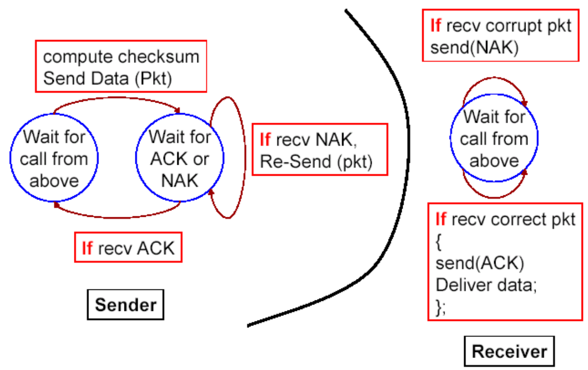


Two scenarios for protocol 4. (a) Normal case. (b) Abnormal case. The notation is (seq, ack, packet number). An asterisk indicates where a network layer accepts a packet.

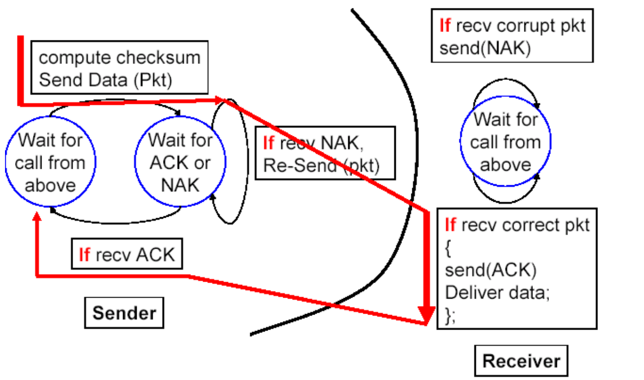
SLLC2.0: Bit Errors

- Underlying channel may flip bits in packet
- How to recover from errors?
  - acknowledgments (ACKs): receiver explicitly tells sender that the packet was received OK
  - negative acknowledgments (NAKs): receiver explicitly tells sender that the packet had errors
  - sender retransmits packet on receiption of NAK
- New mechanisms in SLLC2.0 :
  - error detection (e.g. Checksum)
  - receiver feedback: control messages (ACK,NAK)

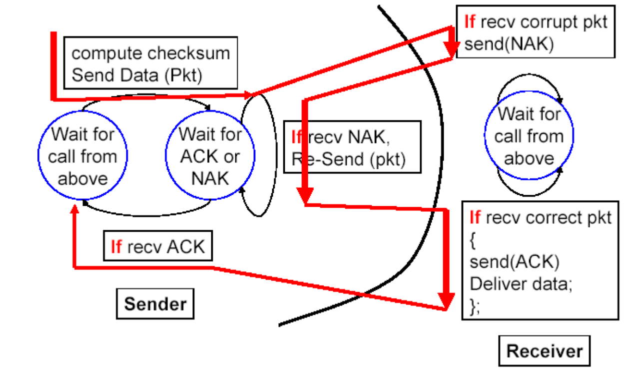
SLLC2.0: FSM specification



SLLC2.0: no errors



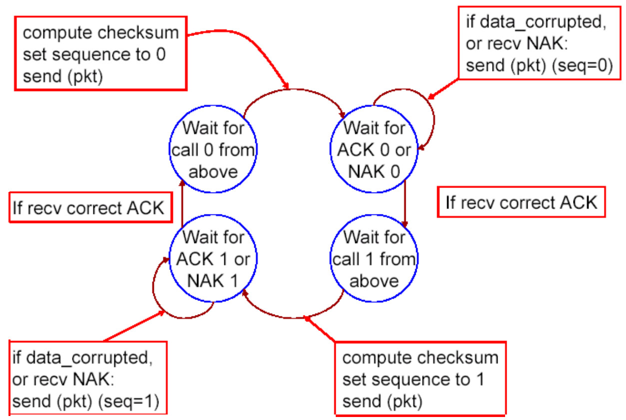
SLLC2.0: Error!



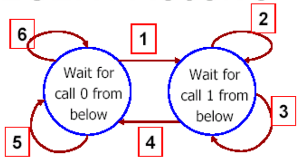
SLLC2.0 Has A Fatal Flaw!

- What happens if ACK/NAK corrupted?
  - Sender doesn't know what happened!
- What to do?
  - Sender ACKs/NAKs receiver's ACK/NAK?
  - Retransmit, but this might cause retransmission of correctly received pkt!
- Handling duplicates:
  - sender adds sequence number to each pkt
  - sender retransmits current pkt if ACK/NAK garbled
  - receiver discards (doesn't deliver up) duplicate pkt

SLLC2.1: handles garbled ACK/NAKs: Sender



SLLC2.1: Receiver



1. IF recv correct, Seq= 0:
{ deliver data; compute checksum; Send ACK pkt }
2. 5. IF recv corrupt:
{compute checksum; Send NAK pkt }
3. IF recv correct, Seq= 0:
{compute checksum; Send ACK pkt }
4. IF recv correct, Seq= 1:
{ deliver data; compute checksum; Send ACK pkt }
6. IF recv correct, Seq= 1:
{compute checksum; Send ACK pkt }

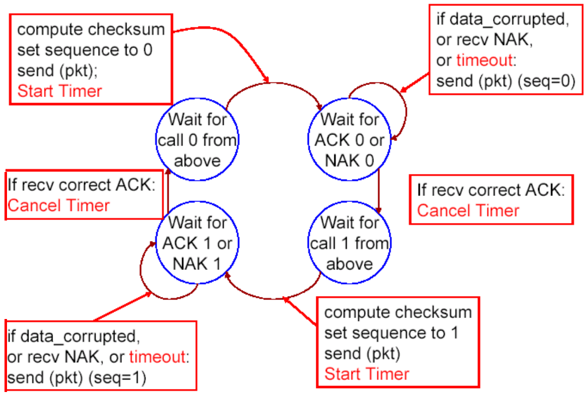
SLLC2.1: Discussion

- Sender:
  - sequence number added to pkt
  - two seq. num's (0,1) will suffice. Why?
  - must check if received ACK/NAK corrupted
  - twice as many states
    - must "remember" whether "current" pkt has 0 or 1 seq. #
- Receiver:
  - must check if received packet is duplicate
    - state indicates whether 0 or 1 is expected pkt seq num
- Note: receiver can not know if its last ACK/NAK received OK at sender

SLLC3.0: With Errors And Loss

- Approach: sender waits "reasonable" amount of time for ACK.
- Retransmits if no ACK received in time.
- If pkt (or ACK) just delayed (not lost):
  - retransmission will be duplicate, but use of sequence number's already handles this
  - receiver must specify sequence number of packet being ACKed
- Requires countdown timer

SLLC3.0 sender



SLLC3.0 In Action

No loss:

Sender | Receiver

send pkt0 → rcv pkt0, send ACK0
rcv ACK0, send pkt1 → rcv pkt1, send ACK1
rcv ACK1, send pkt0 → rcv pkt0, send ACK0

Lost packet:

Sender | Receiver

send pkt0 → rcv pkt0, send ACK0
rcv ACK0, send pkt1 → X (lost)
timeout
resend pkt1 → rcv pkt1, send ACK1
rcv ACK1, send pkt0 → rcv pkt0, send ACK0

SLLC3.0 In Action cont.

Lost ACK:

Sender | Receiver

send pkt0 → rcv pkt0, send ACK0
rcv ACK0, send pkt1 → rcv pkt1, send ACK1 (lost) X
timeout
resend pkt1 → rcv pkt1, send ACK1
rcv ACK1, send pkt0 → rcv pkt0, send ACK0

Premature timeout:

Sender | Receiver

send pkt0 → rcv pkt0, send ACK0
rcv ACK0, send pkt1 → rcv pkt1, send ACK1
timeout, resend pkt1 → rcv pkt1, drop duplicate
rcv ACK1, send pkt0 → rcv pkt0, send ACK0

Performance of SLLC3.0

- SLLC3.0 works, but performance stinks
- example: 1 Gbps link, 15 ms e-e prop. delay, 1KB packet:

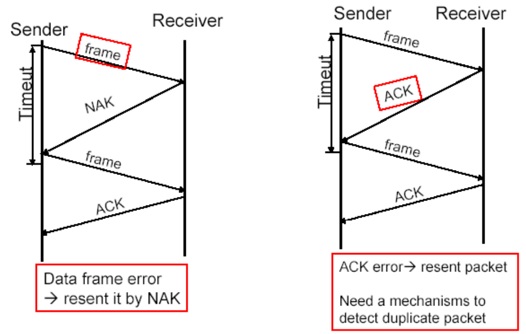
$$T_{transmit} = \frac{8kb/pkt}{10^{**}9\ b/sec} = 8\ \mu s$$

$$\text{Utilization} = U = \frac{\text{fraction of time}}{\text{sender busy sending}} \approx \frac{8\ \mu s}{30\ ms} = 2.6e\text{-}4$$

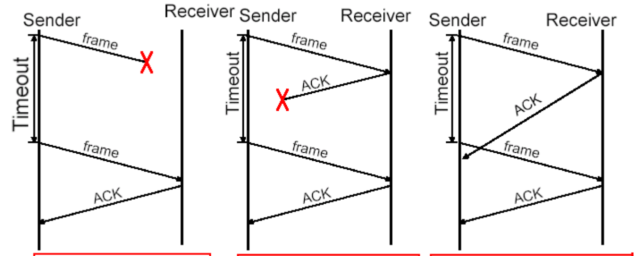
- 1KB pkt every 30 msec -> 266kbps throughput over 1 Gbps link
- network protocol limits use of physical resources!

In fact, SLLC 3.0 is Stop-and-Wait Protocol (SWP), see textbook P159

SWP - Summary
What Can Go Wrong?



SWP - Summary
What Can Go Wrong?



Frame lost → resent it on Timeout

ACK lost → resent packet

Need a mechanisms to detect duplicate packet

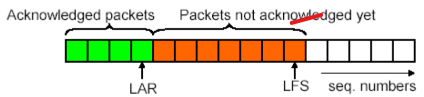
ACK delayed → resent packet
Need a bit to differentiate between ACK for current and previous packet

Solution

- Don't wait for the ACK of the previous packet before sending the next one!

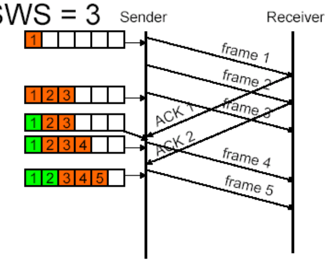
Sliding Window Protocol: Sender

- Each packet has a sequence number
  - Assume infinite sequence numbers for simplicity
- Sender maintains a window of sequence numbers
  - SWS (sender window size) – maximum number of packets that can be sent without receiving an ACK
  - LAR (last ACK received)
  - LFS (last frame sent)



Example

- Assume SWS = 3



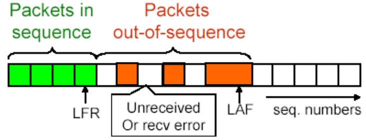
Note: usually ACK contains the sequence number of the first packet in sequence expected by receiver

Sliding Window Protocol: Receiver

- Receiver maintains a window of sequence numbers
  - RWS (receiver window size) – maximum number of out-of-sequence packets that can received
  - LFR (last frame received) – last frame received in sequence
  - LAF (last acceptable frame)
  - LAF – LFR <= RWS

Sliding Window Protocol: Receiver

- Let SeqNum be the sequence number of arriving packet
- If (seqNum <= LFR) or (seqNum >= LAF)
  - Discard packet
- Else
  - Accept packet
  - ACK largest sequence number seqNumToAck, such that all packets with sequence numbers <= seqNumToAck were received (LFR)



SWP is a sliding window protocol, why?

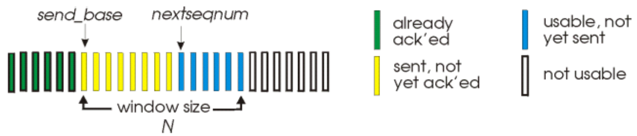
Go Back N (GBN)

Selective Repeat Protocol (SRP)

2 Go-Back-N

Sender:

- k-bit sequence # in packet header
- "window" of up to N, consecutive unack'ed packets allowed



ACK(n): ACKs all packets up to, including sequence # n - "cumulative ACK"

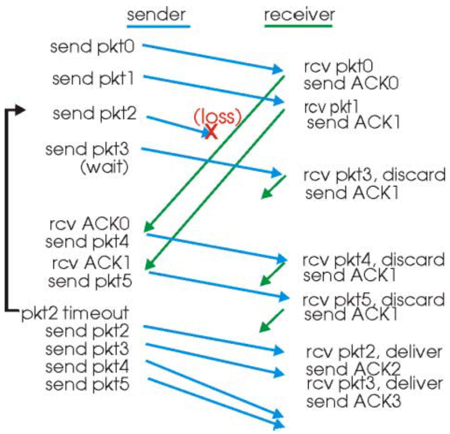
timer for each in-flight packet

timeout(n): retransmit packet n and all higher sequence # packets in window

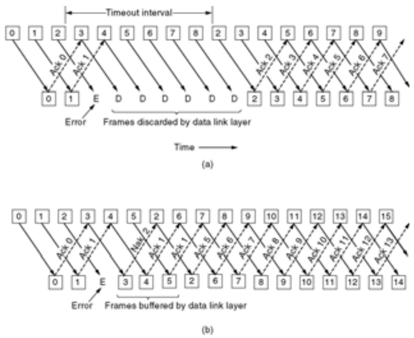
GBN: receiver extended FSM

receiver simple:

- ACK-only: always send ACK for correctly-received packet with highest *in-order* sequence #
  - may generate duplicate ACKs
  - need only remember `expectedseqnum`
- out-of-order pkt:
  - discard (don't buffer) -> no receiver buffering!
  - ACK packet with highest in-order sequence #



A Protocol Using Go Back N



- Pipelining and error recovery. Effect on an error when
- (a) Receiver's window size is 1.
- (b) Receiver's window size is large.

Sliding Window Protocol Using Go Back N

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

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

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

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

  s.info = buffer[frame_nr];             /* insert packet into frame */
  s.seq = frame_nr;                      /* insert sequence number into frame */
  s.ack = (frame_expected + MAX_SEQ) % (MAX_SEQ + 1);/* piggyback ack */
  to_physical_layer(&s);                 /* transmit the frame */
  start_timer(frame_nr);                 /* start the timer running */
}
```

Continued →

Sliding Window Protocol Using Go Back N

```
void protocol5(void)
{
  seq_nr next_frame_to_send;        /* MAX_SEQ > 1; used for outbound stream */
  seq_nr ack_expected;              /* oldest frame as yet unacknowledged */
  seq_nr frame_expected;            /* next frame expected on inbound stream */
  frame r;                          /* scratch variable */
  packet buffer[MAX_SEQ + 1];       /* buffers for the outbound stream */
  seq_nr nbuffered;                 /* # output buffers currently in use */
  seq_nr i;                         /* used to index into the buffer array */
  event_type event;

  enable_network_layer();           /* allow network_layer_ready events */
  ack_expected = 0;                 /* next ack expected inbound */
  next_frame_to_send = 0;           /* next frame going out */
  frame_expected = 0;               /* number of frame expected inbound */
  nbuffered = 0;                    /* initially no packets are buffered */
```

Continued →

Sliding Window Protocol Using Go Back N

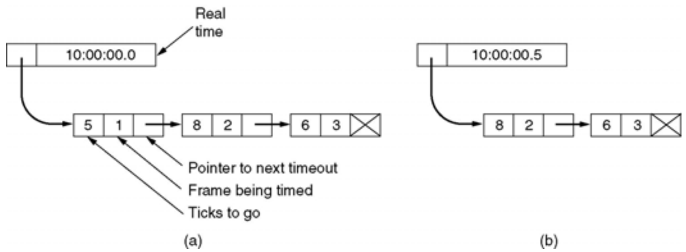
```
  while (true) {
   wait_for_event(&event);          /* four possibilities: see event_type above */

   switch(event) {
     case network_layer_ready:      /* the network layer has a packet to send */
          /* Accept, save, and transmit a new frame. */
          from_network_layer(&buffer[next_frame_to_send]); /* fetch new packet */
          nbuffered = nbuffered + 1;  /* expand the sender's window */
          send_data(next_frame_to_send, frame_expected, buffer);/* transmit the frame */
          inc(next_frame_to_send);  /* advance sender's upper window edge */
          break;

     case frame_arrival:            /* a data or control frame has arrived */
          from_physical_layer(&r);  /* get incoming frame from physical layer */

          if (r.seq == frame_expected) {
               /* Frames are accepted only in order. */
               to_network_layer(&r.info);  /* pass packet to network layer */
               inc(frame_expected);  /* advance lower edge of receiver's window */
          }
```

Continued →

Sliding Window Protocol Using Go Back N

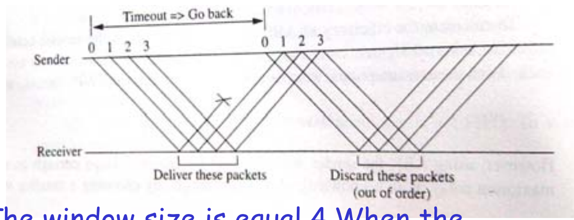
```
          /* Ack n implies n – 1, n – 2, etc.  Check for this. */
          while (between(ack_expected, r.ack, next_frame_to_send)) {
               /* Handle piggybacked ack. */
               nbuffered = nbuffered    1; /* one frame fewer buffered */
               stop_timer(ack_expected); /* frame arrived intact; stop timer */
               inc(ack_expected);      /* contract sender's window */
          }
          break;

     case cksum_err: break;            /* just ignore bad frames */

     case timeout:                     /* trouble; retransmit all outstanding frames */
          next_frame_to_send = ack_expected;    /* start retransmitting here */
          for (i = 1; i <= nbuffered; i++) {
               send_data(next_frame_to_send, frame_expected, buffer);/* resend 1 frame */
               inc(next_frame_to_send);  /* prepare to send the next one */
          }

   }

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

Sliding Window Protocol Using Go Back N (2)



Simulation of multiple timers in software.

GO BACK N



The window size is equal 4.When the acknowledgment of a packet (here pack 0) fails to arrive within a timeout,the sender starts retransmitting that packet and all the subsequent packets.

Selective Repeat

- receiver *individually* acknowledges all correctly received packets
  - buffers packets, as needed, for eventual in-order delivery to upper layer
- sender only resends packets for which ACK not received
  - sender timer for each unACKnowledged packet

Selective repeat

sender

data from above :

- if next available sequence # in window, send packet

timeout(n):

- resend pkt n, restart timer

ACK(n) in [sendbase,sendbase+N]:

- mark packet n as received
- if n smallest unACKed packet, advance window base to next unACKed sequence #

receiver

packet n in [rcvbase, rcvbase+N-1]

- send ACK(n)
- out-of-order: buffer
- in-order: deliver (also deliver buffered, in-order packets), advance window to next not-yet-received packet

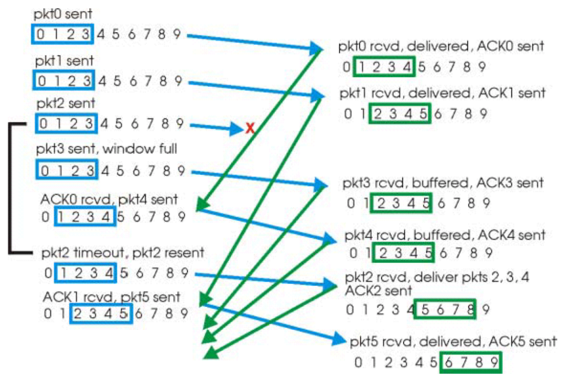
packet n in [rcvbase-N,rcvbase-1]

- ACK(n)

otherwise:

- ignore

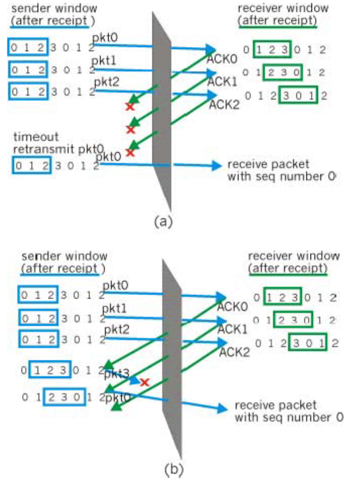
Selective repeat in action



Selective repeat: dilemma

Example:

- seq #'s: 0, 1, 2, 3
- window size=3



- receiver sees no difference in two scenarios!
- incorrectly passes duplicate data as new in (a)

Q: what relationship between seq # size and window size?

A Sliding Window Protocol Using Selective Repeat

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

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

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

static void send_frame(frame_kind fk, seq_nr frame_nr, seq_nr frame_expected, packet buffer[ ])
{
/* Construct and send a data, ack, or nak frame. */
  frame s;                                         /* scratch variable */

  s.kind = fk;                                     /* kind == data, ack, or nak */
  if (fk == data) s.info = buffer[frame_nr % NR_BUFS];
  s.seq = frame_nr;                                /* only meaningful for data frames */
  s.ack = (frame_expected + 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 */
}
```

Continued →

A Sliding Window Protocol Using Selective Repeat (2)

```
void protocol6(void)
{
  seq_nr ack_expected;                 /* lower edge of sender's window */
  seq_nr next_frame_to_send;           /* upper edge of sender's window + 1 */
  seq_nr frame_expected;               /* lower edge of receiver's window */
  seq_nr too_far;                      /* upper edge of receiver's window + 1 */
  int i;                               /* index into buffer pool */
  frame r;                             /* scratch variable */
  packet out_buf[NR_BUFS];             /* buffers for the outbound stream */
  packet in_buf[NR_BUFS];              /* buffers for the inbound stream */
  boolean arrived[NR_BUFS];            /* inbound bit map */
  seq_nr nbuffered;                    /* how many output buffers currently used */
  event_type event;

  enable_network_layer();              /* initialize */
  ack_expected = 0;                    /* next ack expected on the inbound stream */
  next_frame_to_send = 0;              /* number of next outgoing frame */
  frame_expected = 0;
  too_far = NR_BUFS;
  nbuffered = 0;                       /* initially no packets are buffered */
  for (i = 0; i < NR_BUFS; i++) arrived[i] = false;
```

Continued →

A Sliding Window Protocol Using Selective Repeat (3)

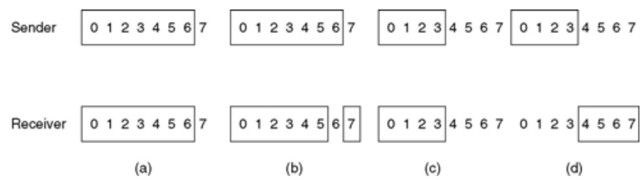
```
while (true) {
  wait_for_event(&event);                      /* five possibilities: see event_type above */
  switch(event) {
    case network_layer_ready:                  /* accept, save, and transmit a new frame */
          nbuffered = nbuffered + 1;           /* expand the window */
          from_network_layer(&out_buf[next_frame_to_send % NR_BUFS]); /* fetch new packet */
          send_frame(data, next_frame_to_send, frame_expected, out_buf);/* transmit the frame */
          inc(next_frame_to_send);             /* advance upper window edge */
          break;

    case frame_arrival:                        /* a data or control frame has arrived */
          from_physical_layer(&r);             /* fetch incoming frame from physical layer */
          if (r.kind == data) {
               /* An undamaged frame has arrived. */
               if ((r.seq != frame_expected) && no_nak)
                  send_frame(nak, 0, frame_expected, out_buf); else start_ack_timer();
               if (between(frame_expected, r.seq, too_far) && (arrived[r.seq%NR_BUFS] == false)) {
                    /* Frames may be accepted in any order. */
                    arrived[r.seq % NR_BUFS] = true;       /* mark buffer as full */
                    in_buf[r.seq % NR_BUFS] = r.info;      /* insert data into buffer */
                    while (arrived[frame_expected % NR_BUFS]) {
                         /* Pass frames and advance window. */
                         to_network_layer(&in_buf[frame_expected % NR_BUFS]);
                         no_nak = true;
                         arrived[frame_expected % NR_BUFS] = false;
                         inc(frame_expected);      /* advance lower edge of receiver's window */
                         inc(too_far);             /* advance upper edge of receiver's window */
                         start_ack_timer();        /* to see if a separate ack is needed */
                    }
               }
          }
```

Continued →

A Sliding Window Protocol Using Selective Repeat (4)

```
            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();
  }
}
```

A Sliding Window Protocol Using Selective Repeat (5)

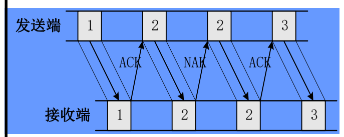


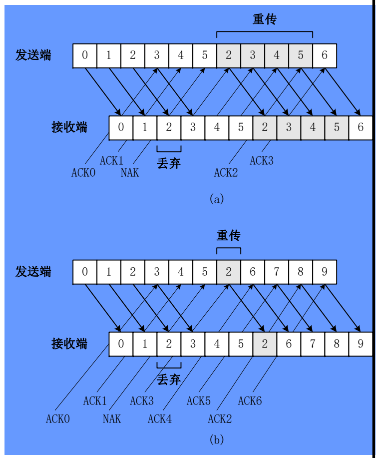
- (a) Initial situation with a window size seven.
- (b) After seven frames sent and received, but not acknowledged.
- (c) Initial situation with a window size of four.
- (d) After four frames sent and received, but not acknowledged.

Comparison

GBN

SWP





SRP

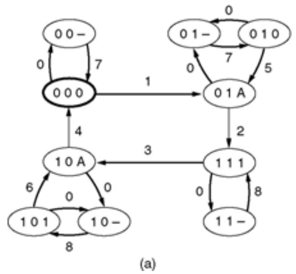
Protocol Analyze

- State diagram
- State table

Protocol Verification

- Finite State Machined Models
- Petri Net Models

Finite State Machined Models

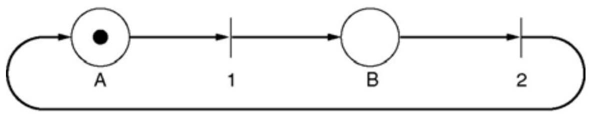


Transition	Who runs?	Frame accepted	Frame emitted	To network layer
0	–	(frame lost)		–
1	R	0	A	Yes
2	S	A	1	–
3	R	1	A	Yes
4	S	A	0	–
5	R	0	A	No
6	R	1	A	No
7	S	(timeout)	0	–
8	S	(timeout)	1	–

(b)

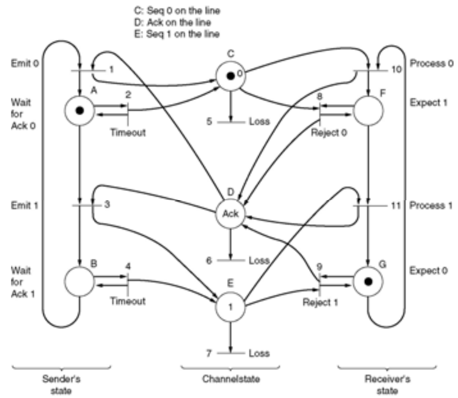
(a) State diagram for protocol 3. (b) Transmissions.

Petri Net Models



A Petri net with two places and two transitions.

Petri Net Models (2)



A Petri net model for protocol 3.

Encapsulation

Format: depends on the protocols but generally consists of three parts:
a header, the packet, a trail
The header identifies the start of the packet and the address in the case of a shared link;
The trainer contains the error control bits .

Representative : HDLC

Example Data Link Protocols

- HDLC – High-Level Data Link Control
- The Data Link Layer in the Internet

HDLC(high-level data link control)

Main concept:

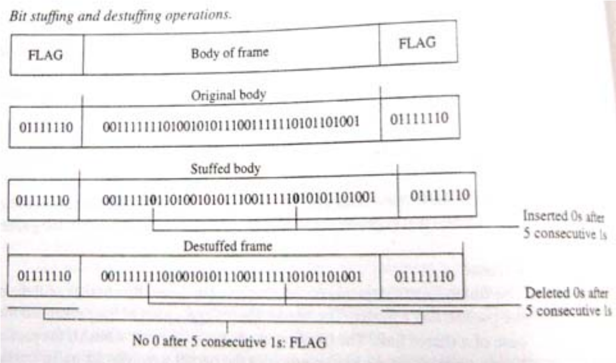
•Station types:
Primary station,
Secondary station,
combined station

•Link structure:
Unbalanced configuration:consists of a primary station and one or more secondary stations;
Balanced configuration: consists of two combined stations

HDLC(high-level data link control)

Main concept:

•Three modes of operation :
NRM:normal response mode
ABM:asynchronous balanced mode
ARM: asynchronous response mode
Extended Modes



High-Level Data Link Control

Bits	8	8	8	≥ 0	16	8
	0 1 1 1 1 1 1 0	Address	Control	Data	Checksum	0 1 1 1 1 1 1 0

Frame format for bit-oriented protocols.

High-Level Data Link Control (2)

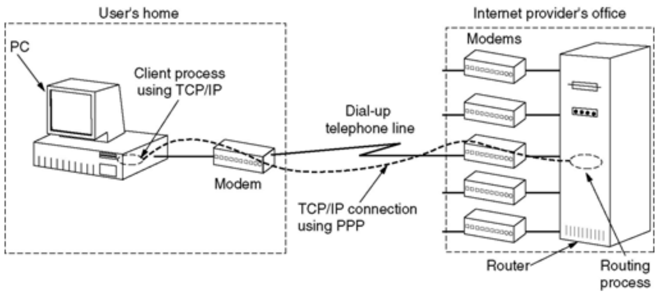
Bits	1		3	1	3
(a)	0	Seq		P/F	Next
(b)	1	0	Type	P/F	Next
(c)	1	1	Type	P/F	Modifier

- Control field of
- (a) An information frame.
- (b) A supervisory frame.
- (c) An unnumbered frame.

The Data Link Layer in the Internet

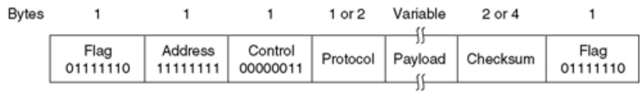
- SLIP(Serial line IP)
- PPP (Point-to-Point protocol)

The Data Link Layer in the Internet



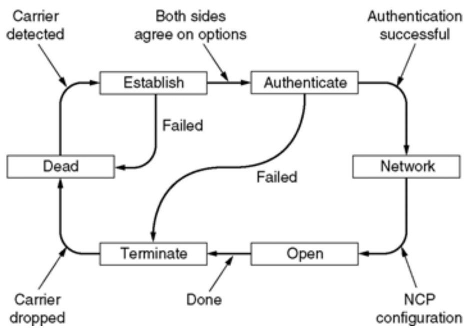
A home personal computer acting as an internet host.

PPP – Point to Point Protocol



The PPP full frame format for unnumbered mode operation.

PPP – Point to Point Protocol (2)



A simplified phase diagram for bring a line up and down.

PPP – Point to Point Protocol (3)

Name	Direction	Description
Configure-request	I → R	List of proposed options and values
Configure-ack	I ← R	All options are accepted
Configure-nak	I ← R	Some options are not accepted
Configure-reject	I ← R	Some options are not negotiable
Terminate-request	I → R	Request to shut the line down
Terminate-ack	I ← R	OK, line shut down
Code-reject	I ← R	Unknown request received
Protocol-reject	I ← R	Unknown protocol requested
Echo-request	I → R	Please send this frame back
Echo-reply	I ← R	Here is the frame back
Discard-request	I → R	Just discard this frame (for testing)

The LCP frame types.

Summary

- There are two steps required to transmit frames (packets) reliable
  - Detect when packets experience errors or are lost
    - Parity
    - Two-dimensional parity
    - Checksum
    - Cyclic Redundancy Check (CRC)
  - Use packet retransmission to recover from errors
    - Stop-and-Wait
    - Sliding window protocol

•The increasing order of complexity and efficiency of the protocols is SWP,ABP,GO BACK N,SRP

Assignment 4 p243 2 5 14