ICOM 5026-090: Computer Networks
Chapter 3: The Data Link Layer

By Dr. Yi Qian
Department of Electronic and Computer Engineering
Fall 2006
Outline

- Design issues
- Error detecting and correcting
- Elementary protocols
- Protocol verification
- Existing protocols
DLL Design Issues

- Services Provided to the Network Layer
- Framing
  - To determine the start and end of a frame
- Error Control
  - To provide reliable communications
- Flow Control
  - To regulate the transmission of sender
    - Example: fast sender and slow receiver
Relationship Between Packets and Frames

Sending machine

Packet

Header Payload field Trailer

Frame

Receiving machine

Packet

Header Payload field Trailer
Services

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

- Unacknowledged connectionless service
- Acknowledged connectionless service
- Acknowledged connection-oriented service
Placement of The Data Link Protocol

Router

Data link layer process

Routing process

Frames here

Packets here

Transmission line to a router

Data link protocol

3

2

2

2

3

UPRM
Framing

- Approaches
  - Character count
  - Flag bytes with byte stuffing
  - Starting and ending flags with bit stuffing
  - Physical layer coding violations
Character Count

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

• Usage
  – Flag (a certain bit pattern) can be used to determine the boundary of a frame

• Problem
  – The bit pattern may appear in the payload

• Solution
  – Use stuffing bits to remove the bit pattern in the payload
    • Character stuffing
    • Bit stuffing
Character Stuffing

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

<table>
<thead>
<tr>
<th>FLAG</th>
<th>Header</th>
<th>Payload field</th>
<th>Trailer</th>
<th>FLAG</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(a)

<table>
<thead>
<tr>
<th>Original characters</th>
<th>After stuffing</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>FLAG</td>
</tr>
<tr>
<td>A</td>
<td>ESC</td>
</tr>
<tr>
<td>A</td>
<td>ESC</td>
</tr>
<tr>
<td>A</td>
<td>ESC</td>
</tr>
</tbody>
</table>

(b)
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.

(a) 0 1 1 0 1 1 1 1 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 1 1 1 1 1 1 1 1 0 1 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 1 1 0 0 1 0
Error Detection and Correction

- Error-Correcting Codes
- Error-Detecting Codes
• Suppose that
  – The original data consists of m bits
  – The number of redundant bits is r
  – The final data that will be transmitted consists of n=m+r bits

• Coding
  – Find a rule to map the m-bit data to n-bit data

• Codeword
  – The sequence of n-bit data
Hamming Distance

• The number of bits that are different in two codewords is called Hamming distance (of the two codewords): d
  – If the distance of two codewords is d, then one codeword can be converted to the other if there are d bit errors

• Example
  – Original data 1: 1000
  – Original data 2: 1001
  – Codeword 1: 10001000
  – Codeword 2: 10011001
  – Parameter: m=4, r=4, n=8
  – Distance: 2
Hamming Distance

• Complete code
  – The set of all valid codewords
  – Note: the total number of possible codewords is $2^n$

• Hamming distance of the complete code
  – The minimum distance between any two different codewords in the complete code
Coding and Hamming Distance

• To detect k errors, the distance of a complete code must be greater than k
  – \( d \geq k+1 \)

• To correct k errors, the distance of a complete code must be greater than 2k
  – \( d \geq 2k+1 \)

• Example
  – All valid codewords:
    • 1111100000, 0000011111, 0000000000, 1111111111
  – Hamming distance: 5
  – Error correction: 2 errors
  – Error detection: 4 errors
Design of Error-Correcting Code

• To correct one error

• Analysis
  – For a valid codeword, there are \( n \) possible codewords, any one of which has a distance \( d=1 \) to the valid codeword
  – To correct one error, these \( n \) codewords are not valid because of the distance
  – Therefore, for one valid codeword, we need at least \( n+1 \) codewords
  – Consequently, we have \( (n+1)2^m \leq 2^n \)

• The requirement can be rewritten as
  – \( m+r+1 \leq 2^r \)
  – This inequity indicates the theoretical lower bound of \( r \)
Hamming Code

• Coding procedure

– Bits that are power of 2 are checking bits
– Other bits are original data
– Each checking bit forces a collection of bits (including itself) to be even (or odd)
– A bit will be checked only by the check bits in its expression of the sum of powers of 2

  • Example: 11=1+2+8, then bit 11 shall be checked in bit 1, 2, and 8
Hamming Code

- Decoding procedure
  - Init a counter to 0
  - For each check bit, if it is not valid, add the index to the counter
  - If, at the end of the procedure, the counter is 0, then there is no error
  - Otherwise, the counter indicates the index of the error bit
Example (even parity)

- Original code: 1 1 0 0 0 0 1
- Add check bits: x x 1 x 1 0 0 x 0 0 1
- Set check bits: 1 0 1 1 1 0 0 1 0 0 1
  
<table>
<thead>
<tr>
<th>bit position</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit position</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
</tbody>
</table>

- If we receive: 1 0 1 1 1 1 0 1 0 0 1

<table>
<thead>
<tr>
<th>bit position</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
</tbody>
</table>
Against Burst Errors

<table>
<thead>
<tr>
<th>Char.</th>
<th>ASCII</th>
<th>Check bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
<td>1001000</td>
<td>00110010000</td>
</tr>
<tr>
<td>a</td>
<td>1100001</td>
<td>10111001001</td>
</tr>
<tr>
<td>m</td>
<td>1101101</td>
<td>11101010101</td>
</tr>
<tr>
<td>m</td>
<td>1101101</td>
<td>11101010101</td>
</tr>
<tr>
<td>i</td>
<td>1101001</td>
<td>01101011001</td>
</tr>
<tr>
<td>n</td>
<td>1101110</td>
<td>01101010110</td>
</tr>
<tr>
<td>g</td>
<td>1100111</td>
<td>01111001111</td>
</tr>
<tr>
<td>c</td>
<td>1100011</td>
<td>10011000000</td>
</tr>
<tr>
<td>o</td>
<td>1101111</td>
<td>11111000011</td>
</tr>
<tr>
<td>d</td>
<td>1100100</td>
<td>10101011111</td>
</tr>
<tr>
<td>e</td>
<td>1100101</td>
<td>00111000101</td>
</tr>
</tbody>
</table>

Order of bit transmission
Error Detecting Code

• Trade-off of error-detection code
  – Less overhead
  – Require retransmission for the whole packet

• Schemes
  – Parity bit(s)
  – Polynomial code or CRC (cyclic redundancy check)
Parity Bit(s)

- Organize the transmission data as an $n \times k$ (i.e. $n$ rows, $k$ columns) matrix
- Use a parity bit for each column
- The data is transmitted one row at a time
- This scheme can detect a single burst of length $n$
Polynomial Code

- Suppose the data consists of k bits
- The k bits are considered as the coefficient of a polynomial
  - \( M(x) = \sum_{j=0}^{k-1} a_j x^j \)
  - The left-most bit is the coefficient of \( x^{k-1} \)
  - The degree of this polynomial is \( k-1 \)
- Example
  - 1, 1, 0, 0, 0, 1 can be expressed as \( x^5 + x^4 + 1 \)
CRC Algorithm

- **General Polynomial**
  - Denoted as $G(x)$, where high and low order bits of the generator must be 1

- **Data**
  - A sequence of bits can be expressed as $M(x)$
  - The length must be greater the $G(x)$

- **Idea of CRC**
  - Append checksum to data such that the polynomial of the frame (data+checksum) is divisible by $G(x)$
Encoding

- Let \( r \) be the degree of \( G(x) \)
- Append \( r \) zeros to the low order of \( m \)-bit data
  - The data is now \( m+r \) bits
  - The output can be expressed as \( x^r M(x) \)
- Divide \( x^r M(x) \) by \( G(x) \) using mod 2 division
- Subtract the remainder from \( x^r M(x) \) using mod 2 subtraction
- The result is the checksumed frame to be transmitted, denoted as \( T(x) \)
Decoding

• At the receiver side, the received frame can be expressed as \([T(x)+E(x)]\)
  – \(E(x)\) represents bit errors

• Calculate \([T(x)+E(x)]/G(x)\)

• Properties
  – All single bit error can be detected if \(G(x)\) contains two or more terms
  – Odd number of bit errors can be detected if \(G(x)\) is divisible by \((x+1)\)
    • Let \(E(x)=(x+1)Q(x)\), then let \(x=1\)
  – A polynomial code with \(r\) check bits can detect all burst errors of length \(<= r\)
Example

Calculation of the polynomial code checksum.

Frame: 1101011011
Generator: 10011
Message after 4 zero bits are appended: 11010110110000

Transmitted frame: 11010110111110
Elementary Data Link Protocols

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

- Consider DLL only
- Node A wants to send data to node B via a reliable, connection-oriented service
- Node A always has packet ready to send
  - A saturated condition
- Machines do not crash
  - We only need to deal with communication errors
- The receiver is always waiting for incoming packets
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 {
    frame_kind kind;
    seq_nr seq;
    seq_nr ack;
    packet info;
} frame; /* frames are transported in this layer */
    /* what kind of a frame is it? */
    /* sequence number */
    /* acknowledgement number */
    /* the network layer packet */

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

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

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

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

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

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

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

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

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

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

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

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

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

- Simplex: data traffic only on one direction
- No transmission error
- Infinite buffer in DLL
- Network layers on both sides are always ready
- Processing time can be ignored
/* 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 */
    }
}

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 */
    }
}
A Simplex Stop-and-Wait Protocol

- **Simplex**: data traffic only on one direction
  - The channel can be half-duplex
- **No transmission error**
- **Finite buffer in DLL**
- **Network layers on both sides are always ready**
- **Processing time cannot be ignored**
  - The sender shall limit its transmission speed
    - **Stop-and-Wait**
/* 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

- Simplex: data traffic only on one direction
- Transmission errors can occur
  - Hardware can detect the error
- Infinite buffer in DLL
- Network layers on both sides are always ready
- Processing time cannot be ignored
  - The sender shall limit its transmission speed
    - Stop-and-Wait
Design Issues

• First step: add a timer at the sender
  – A packet will be retransmitted if the sender does not receive the ACK frame before timeout

• Second step: prevent duplication
  – Use sequence number
  – Question: what is the minimum number of bits for the sequence number?
    • Answer: 1

• Initialization state

• Note: Sender waits for positive acknowledgement before advancing to the next item
  – PAR: Positive Acknowledgement with Retransmission
  – ARQ: Automatic Repeat reQuest
A Simplex Protocol for a Noisy Channel

/* 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;
    frame s;
    packet buffer;
    event_type event;

    next_frame_to_send = 0; /* initialize outbound sequence numbers */
    from_network_layer(&buffer); /* fetch first packet */

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

A positive acknowledgement with retransmission protocol.

Continued →
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);
    if (event == frame_arrival) {
      from_physical_layer(&r);
      if (r.seq == frame_expected) {
        to_network_layer(&r.info);
        inc(frame_expected);
      }
    }
    s.ack = 1 - frame_expected;
    to_physical_layer(&s);
  }
}

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

- **Bidirectional data traffic**
  - We can assume that we have two channels for traffic on both directions
  - ACK number can be “piggyback”ed
    - Consequently, we may be able to save an ACK packet

- **Major concerns**
  - Efficiency
  - Complexity
  - Buffer requirement
Sliding Window

- **Sending window**
  - Determine a set of sequence numbers corresponding to frames that can be sent

- **Receiving window**
  - Determine a set of sequence numbers corresponding to frames that are allowed to be received

- The size of the sending window can be different to the size of the receiving window

- **Buffer size requirement**
  - At the sender side, the buffer size is the same as the sending window
  - At the receiver side, the buffer size equals to the receiving window size
A One-Bit Sliding Window Protocol

- Illustration
- Procedure
- Scenarios
- Problem
Illustration

- 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.
/* 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;
    seq_nr frame_expected;
    frame r, s;
    packet buffer;
    event_type event;

    next_frame_to_send = 0;    /* next frame on the outbound stream */
    frame_expected = 0;        /* frame expected next */
    from_network_layer(&buffer);
    s.info = buffer;
    s.seq = next_frame_to_send;
    s.ack = 1 – frame_expected;
    to_physical_layer(&s);
    start_timer(s.seq);

    /* 0 or 1 only */
    /* 0 or 1 only */
    /* scratch variables */
    /* current packet being sent */

    /* prepare to send the initial frame */
    /* insert sequence number into frame */
    /* piggybacked ack */
    /* transmit the frame */
    /* 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 */
}
Scenarios

- 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.
Problem

• In the previous slide, there are many duplicated frames in scenario B
A Protocol Using Go Back N

- Motivation
  - The round-trip time
- Illustration
- Procedure
- Simulation of Multiple Timers
Illustration

- Pipelining and error recovery. Effect on an error when
  - (a) Receiver’s window size is 1.
  - (b) Receiver’s window size is large.
/* 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\times 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;
    seq_nr ack_expected;
    seq_nr frame_expected;
    frame r;
    packet buffer[MAX_SEQ + 1];
    seq_nr nbuffered;
    seq_nr i;
    event_type event;

    enable_network_layer();
    ack_expected = 0;
    next_frame_to_send = 0;
    frame_expected = 0;
    nbuffered = 0;

    /* MAX_SEQ > 1; used for outbound stream */
    /* oldest frame as yet unacknowledged */
    /* next frame expected on inbound stream */
    /* scratch variable */
    /* buffers for the outbound stream */
    /* # output buffers currently in use */
    /* used to index into the buffer array */

    /* allow network_layer_ready events */
    /* next ack expected inbound */
    /* next frame going out */
    /* number of frame expected inbound */
    /* 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();
}
Simulation of Multiple Timers

- For each unacknowledged frame, we need a timer
- Simulation of multiple timers in software.
A Sliding Window Protocol Using Selective Repeat

- Procedure
- Example
- Maximum 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 */
}
void protocol6(void)
{
    seq_nr ack_expected;
    seq_nr next_frame_to_send;
    seq_nr frame_expected;
    seq_nr too_far;
    int i;
    frame r;
    packet out_buf[NR_BUFS];
    packet in_buf[NR_BUFS];
    boolean arrived[NR_BUFS];
    seq_nr nbuffered;
    event_type event;

    enable_network_layer();
    ack_expected = 0;
    next_frame_to_send = 0;
    frame_expected = 0;
    too_far = NR_BUFS;
    nbuffered = 0;
    for (i = 0; i < NR_BUFS; i++) arrived[i] = false;

    /* lower edge of sender’s window */
    /* upper edge of sender’s window + 1 */
    /* lower edge of receiver’s window */
    /* upper edge of receiver’s window + 1 */
    /* index into buffer pool */
    /* scratch variable */
    /* buffers for the outbound stream */
    /* buffers for the inbound stream */
    /* inbound bit map */
    /* how many output buffers currently used */

    /* initialize */
    /* next ack expected on the inbound stream */
    /* number of next outgoing frame */

    /* initially no packets are buffered */
A Sliding Window Protocol Using Selective Repeat

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;
            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 */
                    }
                }
            }
    }
}
A Sliding Window Protocol Using Selective Repeat

```c
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

- (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.

Sender

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Receiver

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Maximum Window Size

- The maximum receiving window size must be smaller than or equal to the range of sequence number
Protocol Verification

- Finite State Machined Models
- Petri Net Models
State

- Each protocol machine (i.e., sender or receiver) is always in a specific state at every instant of time
  - A state consists of all the values of its variables, including the program counter
- A large number of states can be grouped for purposes of analysis
- Typically, the states are chosen to be those instants that the protocol machine is waiting for the next event to happen
- The total number of states is $2^n$, where $n$ is the number of bits needed to represent all the variables combined
State Transition

- From each state, there are zero or more possible transitions to other states.
- Transitions occur when some event happens:
  - Event
    - A frame is sent
    - A frame arrives
    - A timer expires
    - An interrupt occurs
    - ...

Initial State

- One particular state is designated as the initial state.
- The initial state corresponds to the description of the system when it starts running, or at some convenient starting place shortly thereafter.
- From the initial state, some, perhaps all, of the other states can be reached by a sequence of transitions.
- By using graph theory, it is possible to determine which states are reachable and which are not.
  - This technique is called reachability analysis.
Formal Description

• Formally, a finite state machine model of a protocol can be regarded as a quadruple \((S, M, I, T)\), where:
  – \(S\) is the set of states the processes and channel can be in
  – \(M\) is the set of frames that can be exchanged over the channel
  – \(I\) is the set of initial states of the processes
  – \(T\) is the set of transitions between states
Example

• Protocol
  – A Simplex Protocol for a Noisy Channel

• State
  – Sender
    • Waiting for ACK for frame 0
    • Waiting for ACK for frame 1
  – Receiver
    • Waiting for frame 0
    • Waiting for frame 1
  – Channel
    • Transmitting frame 0
    • Transmitting frame 1
    • Transmitting ACK frame
    • Idle
  – The total number of states is 16
Finite State Machined Models

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

<table>
<thead>
<tr>
<th>Transition</th>
<th>Who runs?</th>
<th>Frame accepted</th>
<th>Frame emitted</th>
<th>To network layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>(frame lost)</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>R</td>
<td>0</td>
<td>A</td>
<td>Yes</td>
</tr>
<tr>
<td>2</td>
<td>S</td>
<td>A</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>R</td>
<td>1</td>
<td>A</td>
<td>Yes</td>
</tr>
<tr>
<td>4</td>
<td>S</td>
<td>A</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>R</td>
<td>0</td>
<td>A</td>
<td>No</td>
</tr>
<tr>
<td>6</td>
<td>R</td>
<td>1</td>
<td>A</td>
<td>No</td>
</tr>
<tr>
<td>7</td>
<td>S</td>
<td>(timeout)</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>S</td>
<td>(timeout)</td>
<td>1</td>
<td>-</td>
</tr>
</tbody>
</table>
Dead Lock

- A deadlock is a situation in which the protocol can make no more forward progress (i.e., deliver packets to the network layer) no matter what sequence of events happens.

- For the graph model, a deadlock is characterized by the existence of a subset of states that is reachable from the initial state and that has two properties:
  - There is no transition out of the subset.
  - There are no transitions in the subset that cause forward progress.
Petri Net Models

• A Petri net has four basic elements:
  – Places
    • Represents a state which (part of) the system may be in
    • Circles
  – Transitions
    • Horizontal or vertical bar
    • Each transition has zero or more input arcs coming from its input places, and zero or more output arcs, going to its output places
  – Arcs
  – Tokens
    • Indicate the current state
    • Heavy dot
Example

- A Petri net with two places and two transitions.
Rules

- A transition is enabled if there is at least one input token in each of its input places.
- Any enabled transition may fire at will, removing one token from each input place and depositing a token in each output place.
- If the number of input arcs and output arcs differs, tokens will not be conserved.
- If two or more transitions are enabled, any one of them may fire.
- The choice of a transition to fire is indeterminate, which is why Petri nets are useful for modeling protocols.
- The Petri net of the previous example is deterministic and can be used to model any two-phase process.
  - E.g., the behavior of a baby: eat, sleep, eat, sleep, and so on.
A Petri net Model for Protocol 3
• Petri nets can be used to detect protocol failures in a way similar to the use of finite state machines.
• The concept of a deadlock in a Petri net is similar to its finite state machine counterpart.
Example Data Link Protocols

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

• SDLC: Synchronous Data Link Control protocol
  – IBM mainframe
  – IBM submit the protocol to ANSI

• ADCCP: Advanced Data Communication Control Procedure
  – ANSI standard

• HDLC: High-level Data Link Control
  – ISO

• LAP: Link Access Procedure
  – Adopted by CCITT (ITU-T), as part of X.25 standard
  – LAPB: improved version
Overview

- Bit oriented
- Bit stuffing
  - FLAG=01111110
- 16-bit CRC
- 8-bit address
- 3-bit sequence number
- Protocols
  - Duplex
    - Piggyback ACK: ACK indicate the first un-received frame
  - Sliding window
    - Go-back-N
    - Selective repeat
Frame Format

- Frame format for bit-oriented protocols.

<table>
<thead>
<tr>
<th>Bits</th>
<th>8</th>
<th>8</th>
<th>8</th>
<th>≥ 0</th>
<th>16</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 1 1 1 1 1 1 0</td>
<td>Address</td>
<td>Control</td>
<td>Data</td>
<td>Checksum</td>
<td>0 1 1 1 1 1 1 0</td>
</tr>
</tbody>
</table>


Control Field

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

<table>
<thead>
<tr>
<th>Bits</th>
<th>1</th>
<th>3</th>
<th>1</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td>0</td>
<td>Seq</td>
<td>P/F</td>
<td>Next</td>
</tr>
<tr>
<td>(b)</td>
<td>1</td>
<td>0</td>
<td>Type</td>
<td>P/F</td>
</tr>
<tr>
<td>(c)</td>
<td>1</td>
<td>1</td>
<td>Type</td>
<td>P/F</td>
</tr>
</tbody>
</table>
Control Field

• **P/F: polling/final**
  – P: invite the peer to send packet
  – F: final
    • Some protocols use this bit to force the peer to send a supervisory frame

• **Frame type**
  – Information frame: normal data frame
  – Supervisory frame:
    • Type 0: ACK frame (continue)
    • Type 1: NACK frame, used for go-back-N
    • Type 2: ACK frame (stop transmitting)
    • Type 3: NACK frame, used for selective repeat
      – The maximum window size must be smaller than or equal to 4
  – Unnumbered frame
    • For control data or connectionless, unreliable service
Unnumbered Frame

- Frame type: 5 bits
- Command
  - DISC: disconnect
  - SNRM: Set Normal Response Mode
    - Reset the sequence number to 0
    - Master-slave mode (asymmetric)
      - One end is master
  - SABM: Set Asynchronous Balanced Mode
    - Used in HDLC and LAPB
    - Reset the sequence number
    - Both sides are equal
  - FRMR: FRaMe Reject
    - Frame is correct (checksum valid), but not valid from the semantics point of view
    - Indicate why the frame was reject
- UA: Unnumbered Acknowledgement
  - For ACK the control frame in a stop-and-wait manner
The Data Link Layer in the Internet

- **Ethernet**
  - Will be discussed in the next chapter

- **Point-to-point connection**
  - **PPP**: Point to Point Protocol
  - Scenarios
    - Router-Router
    - Host-Router
      - Dial-up
PPP Overview

- A home personal computer acting as an internet host.
PPP Overview

- **Standard:** RFC 1661, RFC 1662, RFC 1663
- **Framing**
  - FLAG=01111110
- **Error detection:** CRC (16-bit or 32-bit)
  - Can be negotiated
- **Control protocol**
  - LCP: Link Control Protocol
  - NCP: Network Control Protocol
    - For each supported network layer
- **Retransmission**
  - Not supported as default
  - Defined in RFC 1663 but rarely used
Frame Format

- The PPP full frame format for unnumbered mode operation
- Default length of payload: 1500 Bytes
  - Padding may be required

<table>
<thead>
<tr>
<th>Bytes</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1 or 2</th>
<th>Variable</th>
<th>2 or 4</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flag</td>
<td>01111110</td>
<td>Address</td>
<td>11111111</td>
<td>Control</td>
<td>00000011</td>
<td>Protocol</td>
<td>Payload</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Checksum</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flag</td>
<td>01111110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

UPRM
Phase Diagram

- A simplified phase diagram for bringing a line up and down.

![Phase Diagram](image)
# LCP Frame Types

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