Data Link Layer
Data Link Layer

• Provides a *well-defined service interface* to the network layer.
• Determines how the bits of the physical layer are grouped into frames (*framing*).
• Deals with transmission errors (*CRC* and *ARQ*).
• Regulates the flow of frames.
• Performs general link layer management.
Networks: Data Link Layer

(a) A

Packets

Data link Layer

Physical Layer

---

Packets

Data link Layer

Physical Layer

Frames

(b) A

Medium

B

1 Physical layer entity

2 Data link layer entity

3 Network layer entity

Leon-Garcia & Widjaja: Communication Networks

Networks: Data Link Layer

Figure 5.2
Tanenbaum's Data Link Layer Treatment

• Concerned with communication between two adjacent nodes in the subnet (node to node).

• Assumptions:
  – The bits are delivered in the order sent.
  – **Rigid interface** between the HOST and the node
    ➔ *the communications policy and the Host protocol (with OS effects) can evolve separately.*
  – He uses a simplified model.
Assume the sending Host has *infinite* supply of messages. A node constructs a *frame* from a *single packet* message. The CRC is automatically appended in the hardware. The protocols are developed in increasing complexity to help students understand the data link layer issues.
Basic Elements of ARQ

Packet sequence

Transmitter

Information frames

Receiver

Error-free packet sequence

Station A

Control frames

Station B

Information Frame

Information packet

Header

CRC

CRC

Header

Control frame

Networks: Data Link Layer
Leon-Garcia & Widjaja: Communication Networks

Figure 5.8
Figure 3-9. Some definitions needed in the protocols to follow. These are located in the file protocol.h.
Figure 3-9. 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
/* 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 */
    }
}
/* 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;
    packet buffer;
    event_type event;

    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;
    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 */
    }
}
Ambiguities with Stop-and-Wait

[unnumbered frames]

(a) Frame 1 lost

(b) ACK lost

In parts (a) and (b) transmitting station A acts the same way, but part (b) receiving station B accepts frame 1 twice.
State Machine for Stop-and-Wait

Global State: 
\( (S_{last}, R_{next}) \)

- Error-free frame 0 arrives at receiver
- ACK for frame 0 arrives at transmitter
- Error-free frame 1 arrives at transmitter
- ACK for frame 0 arrives at transmitter

Copyright ©2000 The McGraw Hill Companies

Leon-Garcia & Widjaja: Communication Networks

Networks: Data Link Layer

Figure 5.11
#define MAX_SEQ 1
typedef enum {frame_arrival, cksum_err, timeout} event_type;
#include "protocol.h"

void sender_par (void)
{
    seq_nr next_frame_to_send;
    frame s;
    packet buffer;
    event_type event;
    next_frame_to_send = 0;
    from_network_layer (&buffer);
    while (true)
    {
        s.info = buffer;
        s.seq = next_frame_to_send;
        to_physical_layer (&s);
        start_timer (s.seq);
        wait_for_event(&event);
        if (event == frame_arrival)
        {
            from_network_layer (&buffer);
            inc (next_frame_to_send);
        }
    }
}

Protocol 3
(PAR) Positive ACK
with Retransmission
[Old Tanenbaum Version]
void receiver_par (void)
{
    seq_nr next_frame_to_send;
    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);
            }
            to_physical_layer (&s);
        }
    }
} /* Note – no sequence number on ACK */
PAR [OLD] problem
Ambiguities when ACKs are not numbered

Transmitting station A misinterprets duplicate ACKs
/* 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;
        s.seq = next_frame_to_send;
        to_physical_layer(&s);
        start_timer(s.seq);
        wait_for_event(&event);

        if (event == frame_arrival) {
            from_physical_layer(&s);
            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 */
            }
        }
    }
}

Figure 3-12. A Positive Acknowledgement with Retransmission protocol.
A Simplex Protocol for a Noisy Channel

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) {
            from_physical_layer(&r);    // a valid frame has arrived. */
            if (r.seq == frame_expected) {
                to_network_layer(&r.info);     // go get the newly arrived frame */
                to_network_layer(&s);         // this is what we have been waiting for. */
                inc(frame_expected);       // pass the data to the network layer */
                inc(frame_expected);       // next time expect the other sequence nr */
            }
            s.ack = 1 - frame_expected; // tell which frame is being acked */
            to_physical_layer(&s);    // send acknowledgement */
        }
    }
}

Figure 3-12. A positive acknowledgement with retransmission protocol.
Sliding Window Protocols

[Tanenbaum]

- Must be able to transmit data in both directions.
- Choices for utilization of the reverse channel:
  - mix DATA frames with ACK frames.
  - Piggyback the ACK
    - Receiver waits for DATA traffic in the opposite direction.
    - Use the ACK field in the frame header to send sequence number of frame being ACKed.
  - ➔ better use of the channel capacity.
Sliding Window Protocols

• ACKs introduce a new issue – how long does receiver wait before sending ONLY an ACK frame.
  ➔ Now we need an ACKTimer!!
  ➔ The sender timeout period needs to be set longer.

• The protocol must deal with the premature timeout problem and be “robust” under pathological conditions.
Sliding Window Protocols

Each outbound frame must contain a sequence number. With $n$ bits for the sequence number field, $\text{maxseq} = 2^n - 1$ and the numbers range from 0 to maxseq.

**Sliding window**: the sender has a window of frames and maintains a list of consecutive sequence numbers for frames that it is permitted to send without waiting for ACKs.

The receiver has a window of frames that has space for frames whose sequence numbers are in the range of frame sequence numbers it is permitted to accept.

Note – sending and receiving windows do NOT have to be the same size.

The windows can be fixed size or dynamically growing and shrinking.
Sliding Window Protocols

The Host is oblivious to sliding windows and the message order at the transport layer is maintained.

sender's window :: holds frames sent but not yet ACKed.

– new packets from the Host cause the upper edge inside the sender’s window to be incremented.

– acknowledged frames from the receiver cause the lower edge inside the sender’s window to be incremented.
Sliding Window Protocols

- All frames in the **sender's window** must be saved for possible retransmission and we need one timer per frame in the window.
- If the maximum sender window size is $B$, the sender needs at least $B$ buffers.
- If the **sender's window** gets full (i.e., it reaches the maximum window size), the protocol must shut off the Host (the network layer) until buffers become available.
Sliding Window Protocols

receiver’s window

- Frames received with sequence numbers outside the receiver’s window are not accepted.

- The receiver’s window size is normally static. The set of acceptable sequence numbers is rotated as “acceptable” frames arrive.

If a receiver’s window size = 1, then the protocol only accepts frames in order.

This scheme is referred to as Go Back N.
Selective Repeat :: receiver’s window size > 1.

- The receiver stores all correct frames within the acceptable window range.
- Either the sender times out and resends the missing frame, or
- Selective repeat receiver sends a NACK frame back to the sender.
Choices in ACK Mechanisms

1. The ACK sequence number indicates the last frame successfully received.

   - OR -

2. ACK sequence number indicates the next frame the receiver expects to receive.

Both of these can be strictly individual ACKs or represent cumulative ACKs. Cumulative ACKs is the most common technique.
/* 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);
    s.info = buffer;
    s.seq = next_frame_to_send;
    s.ack = 1 - frame_expected;
    to_physical_layer(&s);
    start_timer(s.seq);

    while (true) {
        wait_for_event(&event); /* frame_arrival, cksum_err, or timeout */
        if (event == frame_arrival) {
            from_physical_layer(&r);
            if (r.seq == frame_expected) {
                to_network_layer(&r.info);
                inc(frame_expected);
            }
        }

        if (r.ack == next_frame_to_send) { /* handle outbound frame stream. */
            stop_timer(r.ack); /* a frame has arrived undamaged. */
            from_network_layer(&buffer); /* go get it */
            to_network_layer(&r.info);
            inc(next_frame_to_send); /* pass packet to network layer */
            inc(next_frame_to_send); /* invert seq number expected next */
            stop_timer(r.ack);
        }

        s.info = buffer;
        s.seq = next_frame_to_send;
        s.ack = 1 - frame_expected;
        to_physical_layer(&s);
        start_timer(s.seq);
    }
}

Figure 3-14. A 1-bit sliding window protocol.
Go Back N

Go-Back-4: 4 frames are outstanding; so go back 4

Out-of-sequence frames

ACKing next frame expected
Go Back N with NAK error recovery

Transmitter goes back to frame 1

Go-Back-7:

<table>
<thead>
<tr>
<th>A</th>
<th>fr</th>
<th>fr</th>
<th>fr</th>
<th>fr</th>
<th>fr</th>
<th>fr</th>
<th>fr</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B</th>
<th>Out-of-sequence frames</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N</td>
</tr>
<tr>
<td>C</td>
<td>A</td>
</tr>
<tr>
<td>K</td>
<td>K</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>K</td>
<td>K</td>
<td>K</td>
<td>K</td>
<td>K</td>
<td>K</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

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

#define MAX_SEQ 7 /* 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;
    seq_nr ack_unexpected;
    seq_nr frame_expected;
    frame r;
    packet buffer[MAX_SEQ + 1];
    seq_nr nbuffed;
    seq_nr i;
    event_type event;
/* 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 frame */
                inc(next_frame_to_send);    /* prepare to send the next one */
            }
    }

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

Figure 3-17. A sliding window protocol using go back n.
Selective Repeat with NAK error recovery

Figure 5.21
Leon-Garcia & Widjaja: Communication Networks Copyright ©2000 The McGraw Hill Companies
/* Protocol 6 (selective repeat) accepts frames out of order but passes packets to the network layer in order. Associated with each outstanding frame is a timer. When the timer expires, only that frame is retransmitted, not all the outstanding frames, as in protocol 5. */

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

boolean no_nak = true;  /* no nak has been sent yet */
seq_nr oldest_frame = MAX_SEQ + 1;  /* initial value is only for the simulator */

static boolean between(seq_nr a, seq_nr b, seq_nr c)
{
    /* Same as between in protocol 5, but shorter and more obscure. */
    return ((a <= b) && (b < c)) || ((c < a) && (a <= b)) || (b < c) && (c < a));
}

static void send_frame(frame_kind fk, seq_nr frame_nr, seq_nr frame_expected, packet buffer[])
{
    /* Construct and send a data, ack, or nak frame. */
    frame s;
    /* scratch variable */
    s.kind = fk;
    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;
    to_physical_layer(&s);
    /* scratch variable */
    if (fk == data) start_timer(frame_nr % NR_BUFS);
    /* transmit the frame */
    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();  /* 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;
    for (i = 0; i < NR_BUFS; i++) arrived[i] = false;  /* initially no packets are buffered */
while (true) {
    wait_for_event(&event);
    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 checksum_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 */
            break;
        } /* end switch(event) */
    } /* end while (true) */
}

Figure 3-19. A sliding window protocol using selective repeat.