Chapter 3

The Data Link Layer

Data Link Layer Design Issues

- Services Provided to the Network Layer
- Framing
- Error Control
- Flow Control
Functions of the Data Link Layer

- Provide service interface to the network layer
- Dealing with transmission errors
- 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.

Placement of the data link protocol.
Framing

(a) Without errors.  (b) With one error.

Framing (2)

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

(a) 0 1 1 0 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 0 1 1 1 1 0 1 1 1 1 0 1 0 0 1 0
(c) 0 1 1 0 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.

Error Detection and Correction

- Error-Correcting Codes
- Error-Detecting Codes
Error-Correcting Codes

Use of a Hamming code to correct burst errors.

Error-Detecting Codes

Calculation of the polynomial code checksum.
Elementary Data Link Protocols

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

Protocol Definitions

```c
#define MAX_PKT 1024
typedef enum [false, true] boolean;
typedef unsigned int seq_nr;
typedef struct {unsigned char data[MAX_PKT];} packet;/* packet definition */
typedef enum {data, ack, nak} frame_kind;
typedef struct {
    frame_kind kind;
    seq_nr seq;
    seq_nr ack;
    packet info;
} frame;
```

Some definitions needed in the protocols to follow. These are located in the file protocol.h.
Protocol Definitions (ctd.)

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

Unrestricted Simplex Protocol

```c
/** 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 typo;
#include "protocol.h"

void sender (void) {
  frame e;
  packet buffer;

  while (true) {
    from_network_layer (&buffer); /* get something to send */
    s.info = buffer;
    to_physical_layer (&s); /* copy it into a bit transmission */
    /* 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 receiver (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);
    to_network_layer (&r.info); /* pass the data to the network layer */
  }
```
Simplex Stop-and-Wait Protocol

A Simplex Protocol for a Noisy Channel

A positive acknowledgement with retransmission protocol.
A Simplex Protocol for a Noisy Channel (ctd.)

```c
void receiver(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, nto);
                inc(frame_expected);
            } else {
                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
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

```c
/* 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;
    frame_expected = 0;
    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);
    /* next frame on the outbound stream */
    /* frame_expected next */
    /* leach a packet from the network layer */
    /* prepare to send the initial frame */
    /* insert sequence number into frame */
    /* piggybacked ack */
    /* transmit the frame */
    /* start the timer running */
```

Continued →
A One-Bit Sliding Window Protocol (ctd.)

```c
while (true) {
    wait_for_event(&event);
    if (event == frame_arrival) {
        from_physical_layer(&i);
        if (i.seq == frame_expected) {
            to_network_layer(&i.no);
            inc(frame_expected);
        }
        if (i.ack == next_frame_to_send) {
            /* handle outbound frame stream. */
            stop_timer(i.ack);
            from_network_layer(&buffer);
            /* fetch new pkt from network layer */
            inc(next_frame_to_send);
            /* invert sender's sequence number */
        }
        s.info = buffer;
        s.seq = next_frame_to_send;
        s.ack = 1 - frame_expected;
        to_physical_layer(&s);
        start_timer(s.seq);
    }
}
```

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.
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 (e.g. fixed in MAX_SEQ)

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
typedef enum [frame_arrival, cloum_err, timeout, network_layer_ready] event_type;
#include "protocol.h"
static boolean between(seq_nr a, seq_nr b, seq_nr x) {
    /* Return true if a < b < c circularly, false otherwise. */
    if (((a <= b) && (b < c)) || ((c < a) && (a <= b)) || ((b < c) && (c < a))){
        return(true);
    }
    return(false);
}
static void send_data(seq_nr frame_nrr, seq_nr seq, frame_expected, packet buffer[]){
    /* Construct and send a data frame. */
    frame s;
    /* scratch variable */
    s.info = buffer[frame_nrr];
    s.seq = frame_nrr;
    s.ack = (frame_expected + MAX_SEQ) % (MAX_SEQ + 1);
    /* transmit the frame */
    start_timer(frame_nrr);
}
void protocol6(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(); /* 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 →

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

```c
/* Ack n implies n - 1, n - 2, etc. Check for this. */
while (between(ack_expected, i.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); /* contrast sender's window */
}
break;

case checksum_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.
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 2  /* should be 2^n - 1 */
#define NR_BUF8S (MAX_SEQ + 1)

typedef enum (frame_arrival, clear_arr, timeout, network_layer_ready, ack_timeout) event_type;

#include "protocol.h"

bool no_rak = true; /* no rak 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 f, seq_nr frame_nr, seq_nr frame_expected, packet buffer[])
{| /* Construct and send a data, ack, or nak frame. */
  frame is;
  /* scratch variable */
  s.kind = f;
  /* kind == data, ack, or nak */
  if (f == data) s.info = buffer[frame_nr % NR_BUF8S];
  e.seq = frame_nr;
  /* only meaningful for data frames */
  e.ack = (frame_expected + MAX_SEQ) % (MAX_SEQ + 1);
  if (f == nak) no_rak = false;
  /* one nak per frame, please */
  to_physical_layer(s); /* transmit the frame */
  if (f == data) start_timer(frame_nr % NR_BUF8S);
  stop_ack_time(); /* no need for separate ack frame */
|

Continued

Continued
A Sliding Window Protocol Using Selective Repeat (3)

```c
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 */
      send_frame_to_send[seq + (% NR_BUFFS)];  /* 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 */
      if (r.kind == data) {
        /* an undamaged frame has arrived */
        if (!((seq == frame_expected) & & no_nak))
          send_frame(nak, 0, frame_expected, out_buf); else start_ack_timer();
        if (between(frame_expected, r.seq, low) & & (arrival[seq] | % NR_BUFFS) == false)) {
          /* frames may be accepted in any order */
          arrival[seq] | % NR_BUFFS = true;  /* mark buffer as full */
          inc(low, seq | % NR_BUFFS) = r.info;  /* invert data into buffer */
          while (arrival[seq] | % NR_BUFFS) {
            /* pass frames and advance window */
            to_network_layer(fig��frame_expected | % NR_BUFFS);  /* to_network_layer */
            no_nak = true;
            arrival[seq] | % NR_BUFFS = false;
            inc(seq | % NR_BUFFS);  /* advance lower edge of receiver’s window */
            inc(seq, low);  /* 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)

```c
if ((r.kind == nak) & & between(seq | % (MAX_SEQ + 1), next_frame_to_send)) {
  send_frame(data, r.ack + 1) | % (MAX_SEQ + 1), frame_expected, out_buf);
}
while (between(seq | % (MAX_SEQ + 1), next_frame_to_send)) {
  nbuffered = nbuffered - 1;  /* handle piggybacked ack */
  stop_timer(ack_counter | % NR_BUFFS);  /* frame arrived intact */
  inc(seq | % (MAX_SEQ + 1)), /* advance lower edge of sender’s window */
  break;
}
break;
case checksum_error:
  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_BUFFS) 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.

Protocol Verification

- Finite State Machined Models
- Petri Net Models
*Finite State Machined Models

(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.

Example Data Link Protocols

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

Frame format for bit-oriented protocols.

*High-Level Data Link Control (2)

Control field of
(a) An information frame.
(b) A supervisory frame.
(c) An unnumbered frame.
The Data Link Layer in the Internet

A home personal computer acting as an internet host.

*PPP – Point to Point Protocol (RFC 1661)

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)**

<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>

The LCP frame types.