Computer Networks

Chapter 03

Maarten van Steen

Vrije Universiteit Amsterdam
Faculty of Science
Department of Computer Science
Room R4.20. Tel: (020) 444 7784

steen@cs.vu.nl

Contents

Data Link Layer

Physical: Describes the transmission of raw bits in terms of mechanical and electrical issues.

Data Link: Describes how a shared communication channel can be accessed, and **how a data frame can be reliably transmitted**.

Network: Describes how routing is to be done. Mostly needed in subnets.

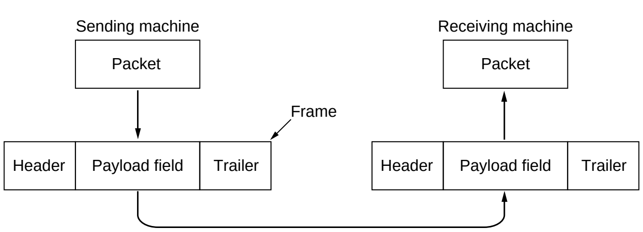
Transport: The hardest one: generally offers connection-oriented as well as connectionless services, and varying degrees of reliability. This layer provides the actual network interface to *applications*.

Application: Contains the stuff that users see: e-mail, remote logins, the Web's exchange protocol, etc.

Note: We'll just concentrate on transmission issues. Channel access is discussed in Chapter 4

Design Issues

- Provide well-defined interface to network layer
- Handle transmission errors
- Regulate flow of data: get sender and receiver in the same pace



Basic Services

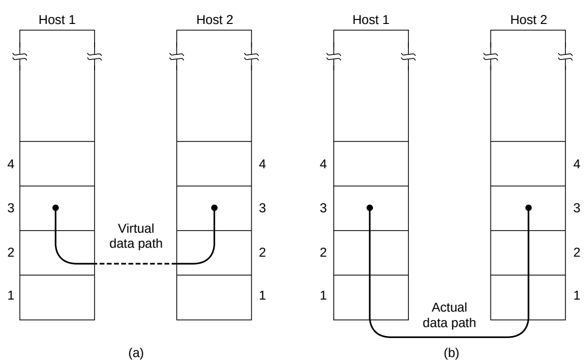
- Network layer passes a number of bits (**frame**) to the data link layer.
- Data link layer is responsible for transmitting the frame to the destination machine.
- Receiving layer passes received frame to its network layer.

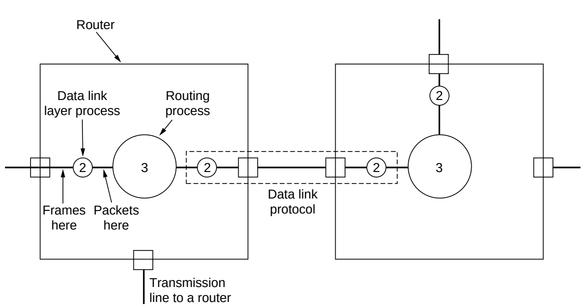
Basic services commonly provided:

- unacknowledged connectionless service (LANs)
- acknowledged connectionless service (Wireless systems).
- acknowledged connection-oriented service (WANs).

Question: Why are we so concerned about error-free frame transmissions? Can't the higher layers take care of that?

Transmission and Routing



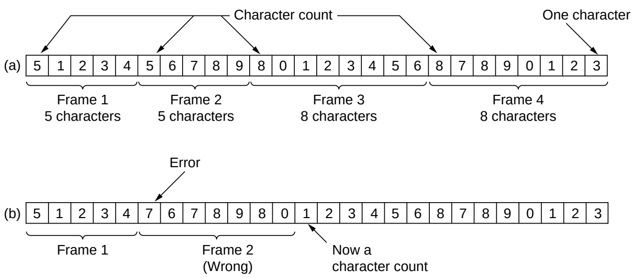


Frames

The physical layer doesn't do much: it just pumps bits from one end to the other. But things may go wrong ⇒ the data link layer needs a means to do retransmissions. The unit of retransmission is a **frame** (which is just a fixed number of bits).

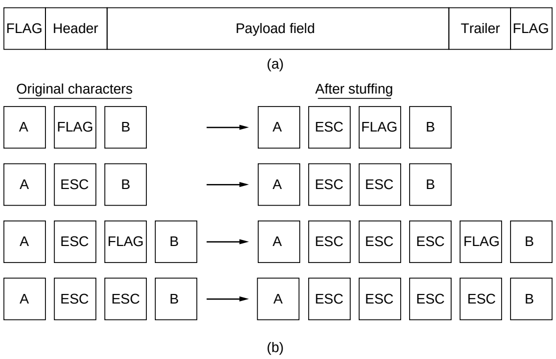
Problem: How can we break up a bit stream into frames?

Counting won't do:



Frames: Stuffing (1/2)

Byte stuffing: Mark the beginning and end of a *byte* frame with two special **flag bytes** – a special bit sequence (e.g. 01111110). If such bytes appear in the original frame, escape them:



Frames: Stuffing (2/2)

Bit stuffing: Escape the flag byte (e.g., 01111110) through an additional bit:

(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

(a) frame to send
(b) frame transmitted over the wire
(c) frame after removing stuffed bits

Error Correction and Detection

Problem: Suppose something went wrong during frame transmission. How do we actually *notice* that something's wrong, and can it be corrected *by the receiver*?

Definition: The Hamming distance between two frames $\mathbf{a}$ and $\mathbf{b}$ is the number of bits at the same position that differ. Example: 10001001 and 10110001 are at Hamming distance 3:

$$\begin{array}{cccccccccc} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & \oplus \\ \hline 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & \end{array}$$

- To *detect* all sets of k or fewer errors, it is necessary and sufficient that the Hamming distance between any two frames is $k+1$ or more.
- To *correct* all sets of k or fewer errors, it is necessary that the Hamming distance between any two frames is $2k+1$ or more.

Error Detection: Parity

Essence: Add a bit to a bit string such that the total number of 1-bits is even (or odd) $\Rightarrow$ the distance between all frames is at least 2.

Conclusion: We can *detect* a single error

Error Correction: Hamming (1/2)

Essence: Every bit at position $2^k, k \geq 0$ is used as a parity bit for those positions to which it contributes:

	b1	b2	b3	b4	b5	b6	b7	b8	b9	b10	b11
1	X		X		X		X		X		X
2		X	X			X	X			X	X
4				X	X	X	X				
8								X	X	X	X

Conclusion: check bit at position 2, is used to even out the bits for positions 2, 3, 6, 7, 10, and 11, so that 1100001 is encoded as (check bits are in boldface):

	b1	**b2**	b3	**b4**	b5	b6	b7	**b8**	b9	b10	b11
1	**X**		X		X		X		X		X
2		**X**	X			X	X			X	X
4				**X**	X	X	X				
8								**X**	X	X	X
	1	**0**	1	**1**	1	0	0	**1**	0	0	1

Error Correction: Hamming (2/2)

Observation: If a check bit at position p is wrong upon receipt, the receiver increments a counter v with p; the value of v will, in the end, give the position of the wrong bit, which should then be swapped.

	b1	**b2**	b3	**b4**	b5	b6	b7	**b8**	b9	b10	b11
1	**X**		X		X		X		X		X
2		**X**	X			X	X			X	X
4				**X**	X	X	X				
8								**X**	X	X	X
C:	**0**	**0**	1	**1**	1	0	0	**0**	1	0	1
S:	**1**	**0**	1	**1**	1	0	0	**1**	0	0	1
R:	**1**	**0**	1	**1**	1	0	0	**1**	1	0	1
F:	**1**	**0**	1	**1**	1	0	0	**1**	0	0	1

S: string sent

R: string received

C: string corrected on the check bits: #1 and #8 corrected $\Rightarrow$ bit #9 is wrong

F: final result after correction

Error Detection: CRC (1/3)

Problem: Error correcting codes are simply too expensive $\Rightarrow$ only use error detection combined with retransmissions.

Example (important): cyclic redundancy check (CRC).

- Associate with a bit string $\mathbf{a} = \langle a_0, a_1, \ldots, a_{m-1} \rangle$ a Boolean polynomial $a(x)$:

$$a(x) = a_0 x^0 + a_1 x^1 + \ldots + a_{m-1} x^{m-1}$$

$$\begin{aligned} \langle 01101 \rangle &\mapsto 0 \cdot x^0 + 1 \cdot x + 1 \cdot x^2 + 0 \cdot x^3 + 1 \cdot x^4 \\ &= x + x^2 + x^4 \end{aligned}$$

- Invent a **generator polynomial**:

$$g(x) = g_0 + g_1 x + \cdots + g_k x^k, \text{ with } g_0 \neq 0 \text{ and } g_k \neq 0$$

Error Detection: CRC (2/3)

- Each bit string $\mathbf{a} = \langle a_0 \ldots a_{m-1} \rangle$ is encoded into a bit string $\mathbf{b} = \langle b_0 \ldots b_{n-1} \rangle$ with $n = m + k$ such that

$$b(x) \;=\; b_0 + b_1 x + \cdots + b_{n-1} x^{n-1} = a(x) \cdot g(x)$$

where all coefficients are calculated using modulo 2 arithmetic.

Example: $\mathbf{a} = \langle 01011 \rangle$

$$\begin{aligned} a(x) &= 0 \cdot x^0 + 1 \cdot x^1 + 0 \cdot x^2 + 1 \cdot x^3 + 1 \cdot x^4 \\ &= x + x^3 + x^4 \\ g(x) &= 1 + x^2 + x^3 \\ a(x) \cdot g(x) &= (x + x^3 + x^4)(1 + x^2 + x^3) \\ &= x + 2x^3 + 2x^4 + x^5 + 2x^6 + x^7 \\ &= x + x^5 + x^7 \end{aligned}$$

Error Detection: CRC (2/3)

After calculating $b(x) = a(x) \cdot g(x)$ send the corresponding bit string $\mathbf{b}$, decode the received string $\hat{\mathbf{b}}$, and construct the quotient and residual:

$$\hat{b}(x) \;=\; \hat{a}(x) \cdot g(x) + \hat{e}(x)$$

$$\begin{aligned} \hat{a}(x) &= \hat{a}_0 + \hat{a}_1 x + \cdots + \hat{a}_{m-1} x^{m-1} \\ \hat{e}(x) &= \hat{e}_0 + \hat{e}_1 x + \cdots + \hat{e}_{k-1} x^{k-1} \end{aligned}$$

Depending on $g(x)$, the number of non-zero coefficients in $\hat{e}(x)$ determine the number of single-bit transmission errors. See Tanenbaum for further details.

Data Link Layer Protocols

- Concentrated on design aspects and error control
- Now: basic protocols and real-world examples

Some basic assumptions:

- We have a machine A that wants to send data to machine B
- There is always enough data for A to send
- There is a well-defined interface to the network layer, and to the physical layer
- The receiver generally waits for an event to happen by calling `wait_for_event`

Unrestricted Simplex Protocol

```
typedef enum {false, true} boolean;
typedef unsigned int seq_nr;
typedef struct {unsigned char data[MAX_PKT];} packet;
typedef enum {data, ack, nak} frame_kind;

typedef struct {
  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;

typedef enum {frame_arrival} event_type;

void sender1(void){
  frame s; packet buffer;
  while (true) {
        from_network_layer(&buffer);
        s.info = buffer;
        to_physical_layer(&s);
  }
}

void receiver1(void){
  frame r; event_type event;
  while (true) {
        wait_for_event(&event);
        from_physical_layer(&r);
        to_network_layer(&r.info);
  }
}
```

Question: What are some of the underlying assumptions here? How does the flow control manifest itself?

Simplex Stop-and-Wait

```
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);
        s.info = buffer;
        to_physical_layer(&s);
        wait_for_event(&event);
  }
}

void receiver2(void){
  frame r, s; event_type event;
  while (true) {
        wait_for_event(&event);
        from_physical_layer(&r);
        to_network_layer(&r.info);
        to_physical_layer(&s);
  }
}
```

Question: What are the assumptions in this case?

Simplex Protocol for Noisy Channel

Let's drop the assumption that the channel is error-free. We do assume that damaged frames can be detected, but also that frames can get lost entirely (how?) ⇒ **Problems:**

- A sender doesn't know whether a frame has made it (correctly) to the receiver. Solution: let the receiver acknowledge undamaged frames.
- Acknowledgments may get lost. Solution: let the sender use a timer by which it simply retransmits unacknowledged frames after some time.
- The receiver cannot distinguish duplicate transmissions. Solution: use sequence numbers.
- Sequence numbers cannot go on forever: we can (and need) only use a few of them. In our example: we need only two (0 & 1).

Simplex Protocol #3 (1/2)

```
#define MAX_SEQ 1
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;
  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_physical_layer(&s);
      if (s.ack == next_frame_to_send) {

        stop_timer(s.ack);
        from_network_layer(&buffer);
        inc(next_frame_to_send);
      }
    }
  }
}
```


Simplex Protocol #3 (2/2)

```
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);
    }
  }
}
```

From Simplex to Duplex

Problem: We want to allow symmetric frame transmission between two communicating parties, rather than transmission in one direction. Don't waste channels, so use the same channel for **duplex** communication.

Principle: Just transmit frames, but distinguish between data, acknowledgments (acks), and possibly negative acks (nacks) in the frame's type field.

Idea: If the other party is going to send data as well, it might as well send acknowledgments along with its data frames ⇒ **piggybacking**.

Question: What's good and bad about piggybacking?

Sliding Windows

Principle: Rather than just sending a single frame at a time, permit the sender to transmit a set of frames, called the **sending window**. A frame is removed from the sending window iff it has been acknowledged.

The receiver has a **receiving window** containing frames it is permitted to receive.

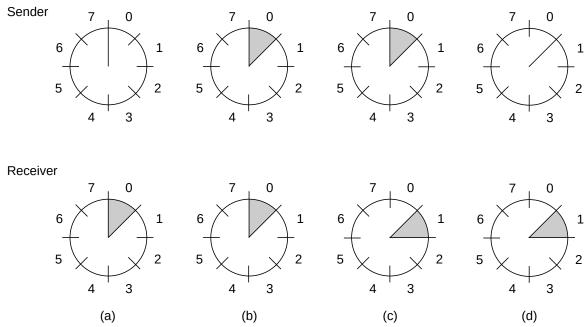
Note: A damaged frame is kept in the receiving window until a correct version is received. Also, frame #N is kept in the window until frame #N-1 has been received (why?)

Note: If both window sizes equal one, we are dealing with the stop-and-wait protocols.

Question: Is there a relationship between window size, transmission time, and propagation time?

Note: Sequence numbers can now correspond to the position a frame has in the window. The receiver needs to keep track of the received frames by their sequence number.

Window Size & Sequence Number



a: Initially.
b: After sending frame #1.
c: After receiving frame #1.
d: After receiving ack for frame #1.

Question: how would you interpret the shaded areas?

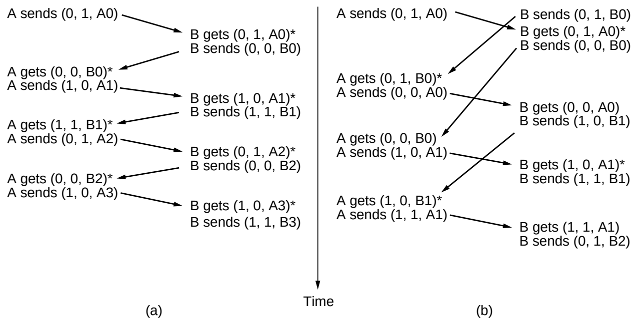
Important: Note that we can stick to an n-bit sequence number $\Rightarrow$ we're not going to run out of sequence numbers.

1-Bit Sliding Window (1/2)

```
void protocol4 (void) {
  seq_nr next_frame_to_send, 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);

  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);
      }
      if (r.ack == next_frame_to_send){
         from_network_layer(&buffer);
         inc(next_frame_to_send);
      }
    }
    s.info = buffer;
    s.seq = next_frame_to_send;
    s.ack = 1 - frame_expected;
    to_physical_layer(&s); start_timer(s.seq);
  }
}
```

1-Bit Sliding Window (2/2)

Observation: All things go well, but behavior is a bit strange when A and B transmit simultaneously:



Observation: We are transmitting more than once, just because the two senders are more or less out of sync.

Error Control (1/3)

Problem: What should the receiver do if a frame is damaged?

- Simply request retransmission of all frames starting from frame #N. If any other frames had been received in the meantime (and stored in the receiver's window), they'll just be ignored ⇒ **go back n**.
- Request just retransmission of the damaged frame, and wait until it comes in before delivering any frames after that ⇒ **selective repeat**.

Error Control (2/3)

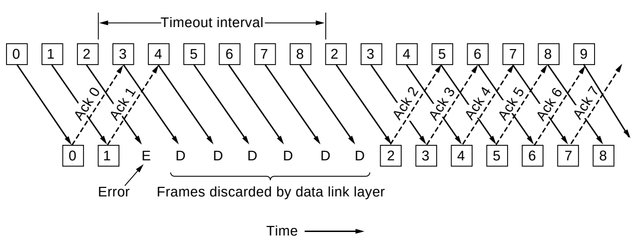
Go-back-N is really simple: the sender keeps a frame in its window until it is acknowledged. If the window is full, the network layer is not allowed to submit new packets.

The receiver hardly needs to keep an account on what happens: if a frame is damaged, its successors in the receive window are ignored.

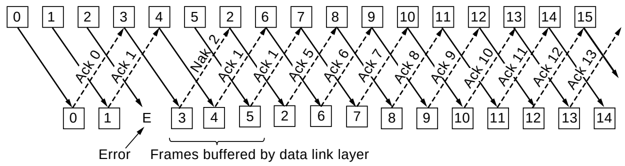
Selective repeat seems to do better because frames aren't discarded, but the administration is much harder.

Error Control (3/3)

The distinction between a window size of 1 and larger:



(a)



(b)

Example: Go-back-n (1/2)

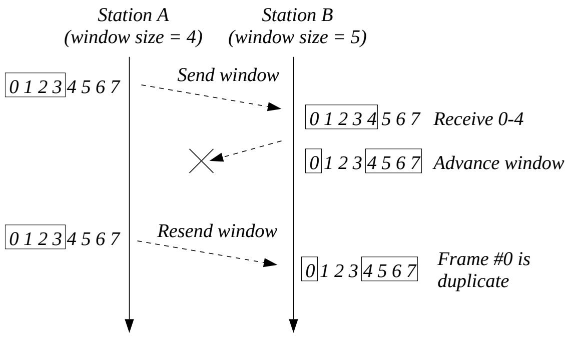
```
#define MAX_SEQ 7 /* should be 2^n - 1 */
typedef enum {frame_arrival, cksum_err,
  timeout, network_layer_ready} event_type;
#include "protocol.h"

static boolean between(seq_nr a, seq_nr b, seq_nr c) {
  /* Return TRUE iff a <= b < c  (cyclic) */
  ...
}

static void send_data(
 seq_nr frame_nr, seq_nr frame_expected, packet buffer[]){
  frame s;
  s.info = buffer[frame_nr]; s.seq = frame_nr;
  s.ack  = (frame_expected + MAX_SEQ) % (MAX_SEQ + 1);
  to_physical_layer(&s); start_timer(frame_nr);
}

void protocol5(void) {
  seq_nr next_frame_to_send, ack_expected, frame_expected;
  frame r;
  packet buffer[MAX_SEQ + 1];
  seq_nr nbuffered, i;
  event_type event;

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

```


Example: Go-back-n (2/2)

```
  while (true) {
    wait_for_event(&event);
    switch(event) {
    case network_layer_ready:
      from_network_layer(&buffer[next_frame_to_send]);
      nbuffered = nbuffered + 1;
      send_data(next_frame_to_send, frame_expected, buffer);
      inc(next_frame_to_send);
      break;

    case frame_arrival:
      from_physical_layer(&r);
      if (r.seq == frame_expected) {
        to_network_layer(&r.info);
        inc(frame_expected);
      }
      while (between(ack_expected, r.ack, next_frame_to_send)) {
        nbuffered = nbuffered - 1;
        stop_timer(ack_expected);
        inc(ack_expected);
      }
      break;

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

    case timeout: /* trouble; retransmit outstanding frames */
      next_frame_to_send = ack_expected;
      for (i = 1; i <= nbuffered; i++) {
        send_data(next_frame_to_send, frame_expected, buffer);
        inc(next_frame_to_send);
      }
    }
    if (nbuffered < MAX_SEQ) enable_network_layer();
    else disable_network_layer();
  }
}
```

Selective Repeat – The Problems

- Frames need not be received in order, i.e. we may have an undamaged frame #N, while still waiting for an undamaged version of #N-1.
- If the receiver delivers all frames in its window just after sending an ack for the entire window, we may have a serious problem:



Solution: we must avoid overlapping send and receive windows ⇒ the highest sequence number must be at least twice the window size.

Example: Selective Repeat (1/3)

```
static boolean between(seq_nr a, seq_nr b, seq_nr c) {...}

static void send_frame(frame_kind fk, seq_nr frame_nr,
       seq_nr frame_expected, packet buffer[]){
  frame s;  s.kind = fk;
  if (fk == data) s.info = buffer[frame_nr % NR_BUFS];
  s.seq = frame_nr;
  s.ack = (frame_expected + MAX_SEQ) % (MAX_SEQ + 1);
  if (fk == nak) no_nak = false;
  to_physical_layer(&s);
  if (fk == data) start_timer(frame_nr % NR_BUFS);
  stop_ack_timer();
}

void protocol6(void){
  seq_nr ack_expected, next_frame_to_send, frame_expected;
  seq_nr nbuffered, too_far; event_type event;
  int i; frame r;
  packet out_buf[NR_BUFS], in_buf[NR_BUFS];
  boolean arrived[NR_BUFS];

  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;
  while (true) {
    wait_for_event(&event);
    switch(event) {
    case network_layer_ready:
      nbuffered = nbuffered + 1;
      from_network_layer(&out_buf[next_frame_to_send % NR_BUFS]);
      send_frame(data,next_frame_to_send,frame_expected,out_buf);
      inc(next_frame_to_send);
      break;
```

Example: Selective Repeat (2/3)

```
    case frame_arrival:
      from_physical_layer(&r);
      if (r.kind == data) {
        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)) {
          arrived[r.seq % NR_BUFS] = true;
          in_buf[r.seq % NR_BUFS] = r.info;
          while (arrived[frame_expected % NR_BUFS]) {
            to_network_layer(&in_buf[frame_expected % NR_BUFS]);
            no_nak = true;
            arrived[frame_expected % NR_BUFS] = false;
            inc(frame_expected);
            inc(too_far);
            start_ack_timer();
          }
        }
      }
      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;
        stop_timer(ack_expected % NR_BUFS);
        inc(ack_expected);
      }
      break;
```


Example: Selective Repeat (3/3)

```
    case cksum_err:
      if (no_nak) send_frame(nak, 0, frame_expected, out_buf);
      break;

    case timeout:
      send_frame(data, oldest_frame, frame_expected, out_buf);
      break;

    case ack_timeout:
      send_frame(ack,0,frame_expected, out_buf);
    }
    if (nbuffered < NR_BUFS) enable_network_layer();
    else disable_network_layer();
  }
}
```

Data Link Layer Protocols

Overview: Take a look how point-to-point connections are supported in, for example, the Internet.

Recall:

- The data link layer is responsible for transmitting **frames** from sender to receiver.
- It can use only the physical layer, which supports only transmission of a bit at a time.
- The DLL has to take into account that transmission errors may occur ⇒ **error control** (ACKs, NACKs, checksums, etc.)
- The DLL has to take into account that sender and receiver may operate at different speeds ⇒ **flow control** (windows, frame numbers, etc.)

High-Level Data Link Control

HDLC: A pretty old, but widely used protocol for point-to-point connections. Is bit-oriented.

Bits	8	8	8	$\geqslant 0$	16	8
	0 1 1 1 1 1 1 0	Address	Control	Data	Checksum	0 1 1 1 1 1 1 0

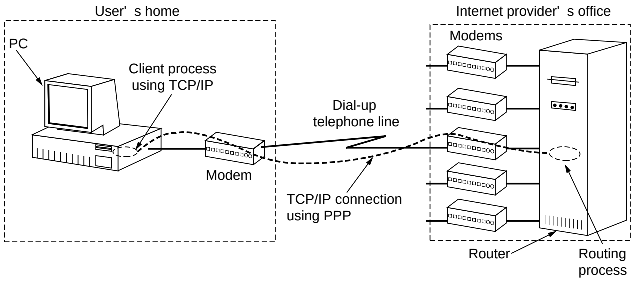
Question: What do we need the *address* field for?

The *control* field is used to dinguish different kinds of frames:

- HDLC uses a sliding window protocol with 3-bit sequencing ⇒ the control field contains sequence numbers, acks, nacks, etc.

Internet Point-to-Point Connections

This is what may happen when you have a simple Internet connection to a provider:



Problem: One way or the other we'd like to use the Internet protocol stack at our home. The bottom line is that we'll have to transfer IP (network) packets across our dial-up line.

Issue: How can we (1) embed IP packets into frames, that can be (2) unpacked at the other end, to be handed to the network layer?

PPP: Point-to-Point Protocol

PPP is the data link protocol for point-to-point connections for the future (with respect to the Internet):

- Proper framing, i.e. the start and end of a frame can be unambiguously detected.
- A separate protocol for controlling the line (setup, testing, negotiating options, and tear-down) (**LCP**)
- Supports a lot of different network layer protocols, not just IP.
- There's no need for fixed network addresses.

The default frame:

Bytes	1	1	1	1 or 2	Variable	2 or 4	1
	Flag 01111110	Address 11111111	Control 00000011	Protocol	Payload	Checksum	Flag 01111110

PPP – Example (1/2)

Suppose you want to set up a true Internet connection to your provider.

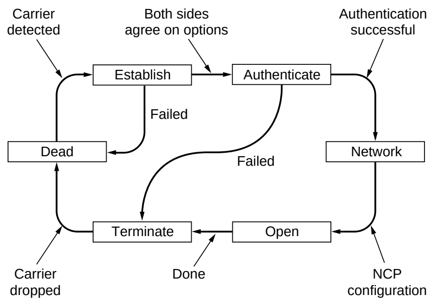
1. Call the provider's router through a modem to set up a physical connection.

2. Your PC starts sending a number of Link Control Packets (LCP) to negotiate what kind of PPP connection you want. Note that these packets are embedded in PPP frames:

   - The maximum payload size in data frames
   - Do authentication (e.g. ask for a password)
   - Monitor the quality of the link (e.g. how many frames didn't come through).
   - Compress headers (useful for slow links between fast computers)

3. Then, we negotiate network layer stuff, like getting an IP address that the provider's router can use to forward packets to you.

PPP – Example (2/3)



Name	Dir.	Description
Confi gure-request	I $\rightarrow$ R	Proposed options and values
Confi gure-ack	I $\leftarrow$ R	All options are accepted
Confi gure-nack	I $\leftarrow$ R	Some options are not accepted
Confi gure-reject	I $\leftarrow$ R	Some options are not negotiable
Terminate-request	I $\rightarrow$ R	Request to shut the line down
Terminate-ack	I $\leftarrow$ R	OK, line shut down
Code-reject	I $\leftarrow$ R	Unknown request received
Protocol-reject	I $\leftarrow$ R	Unknown protocol requested
Echo-request	I $\rightarrow$ R	Please send this frame back
Echo-reply	I $\leftarrow$ R	Here is the frame back
Discard-request	I $\rightarrow$ R	Just discard this frame (testing)

PPP – Example (3/3)

Question: If an IP address is dynamically assigned,

(1) Who does the assignment

(2) Can someone else ever send you data (they don't know your address, do they?)