Data Link Layer

Overview:

- What does the data link layer do?
- How to handle transmission errors.
- Basic transmission protocols, ranging from very simple to hard ones.

The General Layers

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 next time.

Basic Services

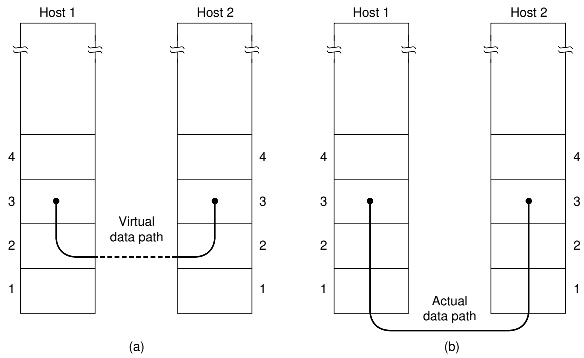
- Network layer passes a number of bits (**frame**) to the data link layer.
- Data link layer is responsible for transmitting the frame to the destination machine.
- Receiving layer passes received frame to its network layer.

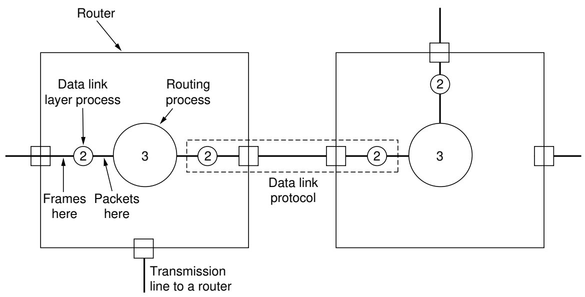
Basic services commonly provided:

- 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



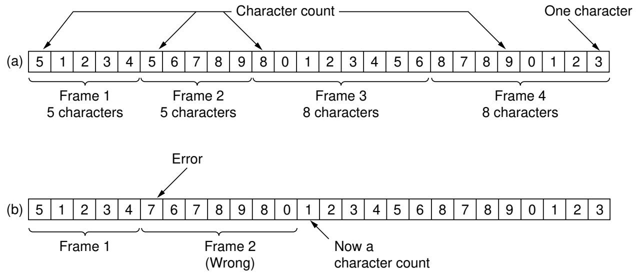


Frames

The physical layer doesn't do much: it just pumps bits from one end to the other. But things may go wrong $\Rightarrow$ 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).

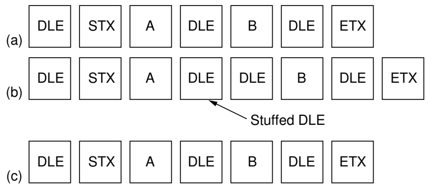
Problem: How can we break up a bit stream into frames?

Counting won't do:

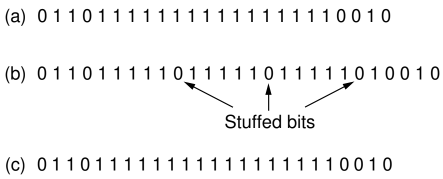


Frames – stuffing

Character stuffing: Mark the beginning and end of a *character* frame with two special escape characters. If such characters appear in the original frame, "escape" them again:



Bit stuffing: Mark the beginning and end of a frame with a **flag byte** – a special bit sequence (e.g. 01111110). Escape the flag byte through an additional bit:



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 **a** and **b** is the number of bits at the same position that differ. Example: 10001001 and 10110001 are at Hamming distance 3:

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

1. 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.
2. 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 – CRC

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**:

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

- 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.

Error Detection – CRC II

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

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

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.

Flow Control

Problem: Suppose that a sender just starts transmitting frames, but that the receiver is too slow to handle the frames as they come in?

Example: If I connect my high-speed Sun Sparcstation to a Intel 386 file server shared by a lot of users, the file server will not be able to handle all my transmitted frames: it simply does not have the processing power.

One way or the other the sender and receiver need to negotiate implicitly or explicitly the **transmission rate**.

Solution: Let the receiver inform the sender how far it has processed the incoming frames. If the receiver is slow, the sender lowers the transmission rate. If the receiver is fast, more frames can be sent.

This requires some kind of **feedback mechanism**.

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).

See Tanenbaum for coding example (do it!)

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.

Principal: 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

Principal: 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.

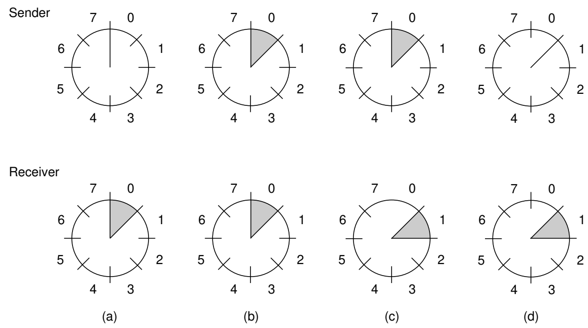
Note: A damaged frame is kept in the receiving window until a correct version is received. Also, frame #N is kept in the window until frame #N-1 has been received (why?)

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 vs 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 ⇒ we're not going to run out of sequence numbers.

Example: 1-Bit Sliding Window

```
void protocol4(void)
{
  ...
  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);
  }
}
```


Error control

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**.

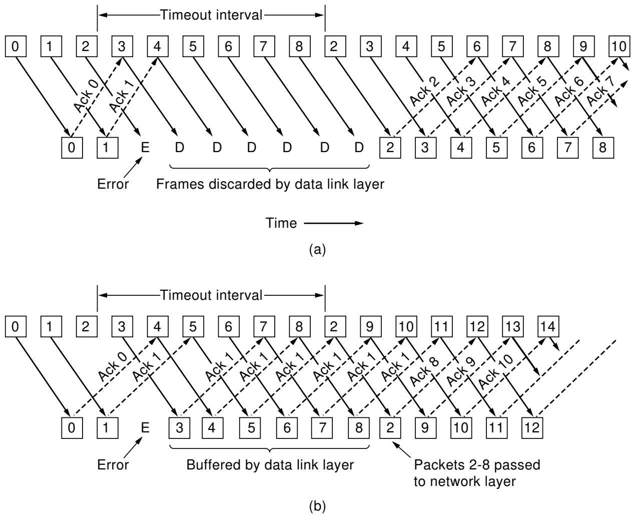
Go-back-N is really simple: the sender keeps a frame in its window until it is acknowledged. If the window is full, the network layer is not allowed to submit new packets.

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.

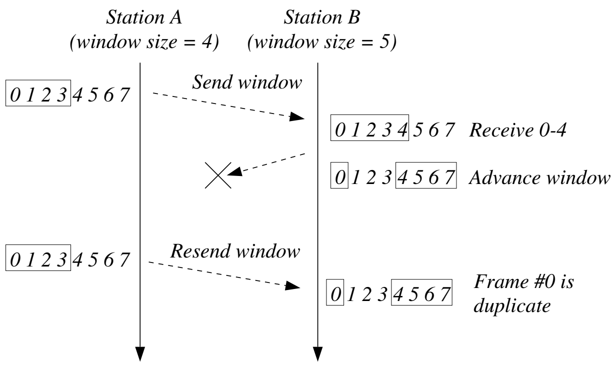
Error control II

The distinction between a window size of 1 and larger:



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.

Data Link Layer Protocols

Overview: Take a look how point-to-point connections are supported in the Internet, and in ATM networks.

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	⩾ 0	16	8
	0 1 1 1 1 1 1 0	Address	Control	Data	Checksum	0 1 1 1 1 1 1 0

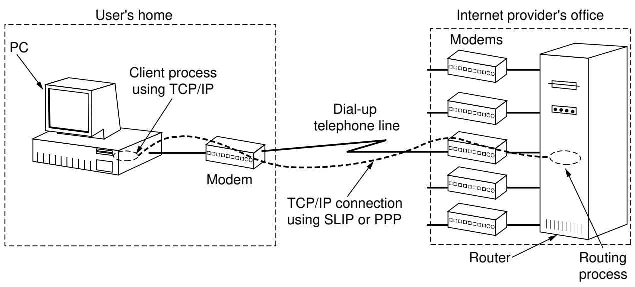
Question: what do we need the *address* field for?

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

- Are we talking about data frames, or frames used for flow and error control?
- 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 (or the VU):



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?

Serial Line IP

SLIP is really simple:

- IP packets are sent raw across the wire, only ended by a special flag byte. **Question**: does this mean that a frame has almost the same layout as an IP packet?
- The flag byte can be escaped (character stuffing) when it occurs in the IP packet.

Problems:

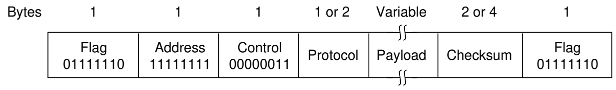
- SLIP requires that both ends have an IP address. Pretty bad (why?).
- There's no error control, so the network layer will have to fix that. Not really an ideal solution (why not?)
- SLIP only supports IP. Connecting to a Novell Network is impossible.
- There's no authentication, so you really don't know whom you're talking to.
- Really great: it's not a standard.

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:



PPP – Example

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.

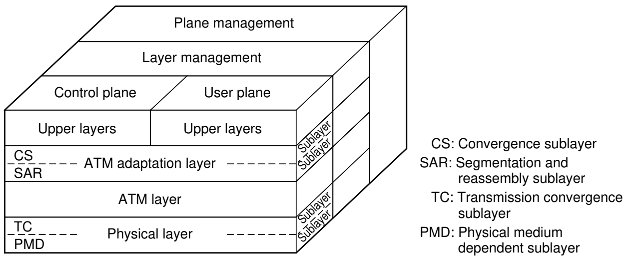
Question: If an IP address is dynamically assigned, (1) who does the assignment, and (2) can someone else ever send you data (they don't know your address, do they?)

DLL and ATM

Recall: ATM is being used as a *switching technology* to build computer networks.

ATM switches, and the computers connected to them, exchange frames which are called **cells**. Each cell has a 5-byte header, and a 48-byte payload field.

Cells are transmitted in "raw" mode across the physical connections between ATM switches. The layering is as follows:



Question: Where does the ATM Data Link Layer fall into place?

ATM DLL I

Sending cells:

- The TC layer takes the header of a cell, of which only 4 bytes contain real information. The 5th byte is used to checksum the others.
- The payload field of an ATM cell (48 bytes) are not checksummed. What's the assumption here?
- The cell is sent out on the wire, a bit at a time. With synchronous communication (i.e. a receiver is *always* expecting something at prefixed times), idle cells are transmitted when necessary.

ATM DLL II

Receiving cells:

- The TC layer just gets incoming *bits*: cells are not delineated $\Rightarrow$ how do we know that a complete cell has been received?
- The TC layer uses a 40-bit shift register, and computes a checksum over the first 32 bits whenever a bit comes in. If it matches the last 8 bits, it assumes it has a cell header.
- When a header is detected, it just passes the contents of the shift register and the next $48 \cdot 8$ bits to the layer above.

Problem: with only 8 bits checksum, we'll always have a $1/256$ chance of getting a correct checksum $\Rightarrow$ repeat the procedure a couple of times.

Note: The TC layer is making use of things that should only be known to the layer above (what?)