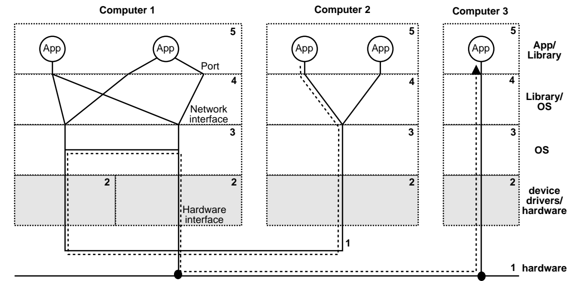
Lecture 4

Data link

References:

- CN 3.1–3.4, 3.6.
- TIv1 2.1–2.6.
- RFC 894 (IP in ethernet), 1042 (IP in IEEE 802), 1661 (PPP and LCP), 1662 (PPP framing), 1663 (PPP with flow control), 1332 (NCP).

Review: Location and functionality



Framing: signal start and end of each unit of transfer, "frame", in the sender, and detect that in the receiver.

Error control: Detect and possibly correct errors in frames.

Flow control: slow down fast sender to let receiver catch up.

Pre-requisites

Framing

- The underlying reason for framing: **economy of scale**.
- By grouping hundreds of bytes into a group, we can associate control information like checksum, data type, destination address, etc., with each group rather than each byte, so reducing their cost.
  They are needed for error correction and flow control, among other things.
- The physical layer communicate a **simple bit stream**, while the network layer transfer in groups called **network packet**.
- How to denote start and end of communication? Easy: interpret the bit stream as bytes, and **send the frame length** before the frame.
- But that **doesn't work**: if the **frame length is corrupted**, sender and receiver might never get **synchronized** again because receiver will never know where is the start of the next message.
  Lesson: length is good only if channel is reliable.

Byte stuffing

- One simple solution: **reserve a byte** for signaling start and end of message (FLAG byte):

- The FLAG byte never appears within the frame. When a packet contains it, the data link layer convert it to 2 bytes: an **escape** followed by a code to denote it. And escape is similarly converted, or **byte-stuffed**.
  Just like that we write `\n` and `\\` in a C++ programs.
- Upper layer will not see it: it is decoded by the data link layer **before** forwarding to the network layer.
  In fact, it is decoded even earlier: before error detection take place.
- There is one restriction, though: this requires the communication channel to be byte-oriented. What if we are **1-bit out-of-sync**?
- A modification of the scheme will remove that restriction.

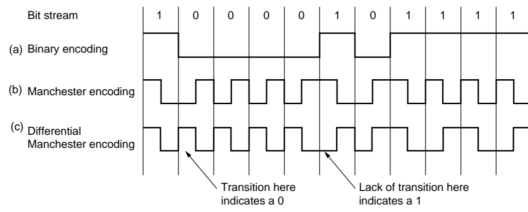
Bit stuffing

- A **special bit pattern**, e.g., 01111110, is used as FLAG.
  This is the value used by PPP.
- Within a frame, everytime we send 5 1-bits, **an extra 0-bit is sent**.
  E.g., if we want to send 0111111100, we instead send 01111101100.
- Upon receiving a sequence of 0111110, the receiver **drops the last 0**.
- Upon receiving a sequence of 0111111, the receiver **drops the next 0** and declare a frame boundary.
  What to do if the next bit is not 0? The data link layer simply declare an error. Normally it would drop the frame, and wait for the FLAG again.

This also frees the sequence of **many 1's**. PPP use this sequence to denote that nothing need to be sent in the upper layer.
So the first 0-bit marks the desire to send something.

Manchester and related encoding

- Sometimes the **physical** layer has **redundancy**, and we can exploit it to avoid paying redundancy twice.
- E.g., most Ethernet implementations uses **manchester encoding**.
  This avoids d.c. (thus save power), make synchronization very easy, and detects most errors, but doubles the bandwidth consumption.



- In manchester encoding, High-low is 1, low-high is 0. We can use high-high and low-low for framing purpose without the need of stuffing.

Error control

- Manchester encoding provide a very expensive form of error control, i.e., use half the bandwidth. We want something cheaper.
- Since noise is rare and bursty, we can use a few bits to detect errors in many bits for more efficient error detection.
- A simple example is **odd-parity**. The idea: at the end of every some (say 8) bits, an extra bit is added, in a way that the number of 1-bits of the result is odd. This detects an error if **1 of the 9 bits** has an error.
  One can use even-parity as well, with similar behaviour.
- But if 2 bits have error, the error goes undetected. It is also quite expensive: more than 10% of the bandwidth is used for error detection.
  In fact, if an even number of bits have error, it goes undetected.
- An alternative: **checksum**. **Sum up all the bytes**, and append the **least-significant** byte of the sum to the frame.
- Better, but still, byte reversals are not detected.

Cyclic Redundancy Code

- The idea of parity is to **append a bit** so that the **result has certain properties** for checking. This idea can be extended.
- Instead of appending a bit, we can **append a code of many bits**.
- One idea is to make sure the result is **divisible by a certain number (generator)**. But this doesn't match well with a bitwise system.
  E.g., if each word is 16-bit, using a generator 65537 => some data has no code that works, using generator 65535 => the code 65535 is wasted, and using generator 65536 => only the code 0 is used.
- A simple modification can fix this: do the division normally, but **never borrow on subtraction**. Instead, all carry and borrow are silently ignored.
  We say we are performing a modulo-2 polynomial division.
- This has an addition advantage: everything can be **done very quickly by hardware**, using just a bit shifter together with an XOR logic.
  So most network interface has hardware to do CRC by itself.

CRC Example

Check divisibility of 11010110111110 (message 1101011011, checksum 1110) by generator 10011.

```
        1100001010
10011 | 11010110111110
        10011
         10011
         10011
              10111
              10011
                10011
                10011
```

Remainder is 0, so we conclude it is divisible.

How to generate CRC at the first place? Simple: do the same thing, assuming checksum being 0. The resulting remainder is the real checksum.

CRC Guarantees

Why this strategy? If the generator is chosen carefully, CRC provides some nice guarantee. For a $r + 1$ bit generator (i.e., r checksum bits),

- Average detection rate as predicted by simple probability: $1/2^r$.
- Suppose the **last bit of generator is 1**. If a message contains **one burst error** of $n \le r$ bits, it is always detected.
- Suppose **there are even number of bits in generator**. If a message an odd number of bits has error, it is always detected.
- Some generators can guarantee errors of 2 bursts of 1-bit errors will be detected if the bursts are **not separated too far apart**.
  E.g., the generator 1100000000000001 detects such errors up to a separation of 32768.

The standard of data link layer defines what generator to use, which is chosen to take advantages of these.

Basis in coding theory

For really noisy channels, or when resending is very costly, just detecting an error is not enough: it has to be **corrected**.

- Like error detection, we will **increase the number of possible code** (by adding bits) and use **only a tiny fraction of it**.
- Suppose we want to **correct all 1-bit errors** of an n bits code. Then all the n 1-bit modifications of a valid code must not collide with any other valid code or 1-bit modifications of them.
- E.g., if a valid code is 10011, then the code 10101 must be invalid, because 10111 is a 1-bit modification of both codes.
- In other words, **codes must differ by at least 3 bits**. Each valid code "reserves" $n + 1$ codes of the 2^n possible codes. So **no more than** $2^n/(n + 1)$ **codes are usable**.
- Can we achieve optimality? Yes: hamming code.

Hamming code

Some nice ideas is best understood from the result. Hamming code is one example: it is clear once we see the scheme.

- Suppose the code has $n - 1$ bits, where n is a power of 2. We will call them bit 1 to bit $n - 1$.
- Bit 1, 2, 4, 8, … are used to hold **parity bits** (for error correction), while all **other bits are for data**.
  For our example, we use even parity. Odd parity works as well.
- Each parity bit keeps the parity of **some bits**: bit 1 keeps parity of bits 1, 3, 5, 7, …; bit 2 keeps parity of bits 2, 3, 6, 7, …
- In general, each data bit (say, bit 9) has the **bit number** (9) written as binary (1001), i.e., sum of powers of 2 (i.e., 1+8). Then it is included in those parity bits (1 and 8).

Example

Suppose we want to encode 1010. The code is then ??1?010.
We count bits from the left. Note that 3 parity bits is used to send 4 bits, which is quite costly. But 4 parity bits can send 11 bits, 5 parity bits can send 26 bits.

- Parity of bit 1: ?, 1, 0, 0; so ? = 1.
- Parity of bit 2: ?, 1, 0, 1; so ? = 0.
- Parity of bit 4: ?, 0, 1, 0; so ? = 1.

The full code to send is 1011010. Suppose a bit is affected by a 1-bit error, causing the word 1011110 to be received, affecting bit 5.

When the receiving end checks for parity, it finds that parities involving bit 1 and bit 4, are wrong.

Add up those parity bit numbers, and we know which bit is wrong!
Since that is the only bit which can affect those sequence of parity bits.

Protocols

Data link assumptions and requirements

Now we are ready to explain the actual dialog. Before we go into actual protocols, let see what we want.

Assumption: what data link needs from physical layer

- Physical layer **never re-order frames**.
- Transmission **delay** is nearly **constant**.

Requirements: what is provided to network layer

- Transfer speed: it should utilize physical layer channel well.
- If there is an error in the physical link, it **will be detected** or corrected using checksum, hamming code, whatever.
- Also desired—resend dropped frames, suppress duplicated frames, don't introduce re-ordering.

Simple protocol without flow control

- Simplest protocol:
  - **Sender**: the data link layer of the sender repeatedly get a network packet, compute the checksum and perform stuffing to create frames, and send it down to physical layer.
  - **Receiver**: unstuff the frame, check the checksum. If it matches, give network layer the packet. Otherwise, tell it you get an error frame.
- Will this work? Yes, if the network layer always has time to work with the frame.
- What if not? Sooner or later the **datalink of receiver will use up its buffer space** sooner or later.
  Then some frames get dropped and needs to be resent.

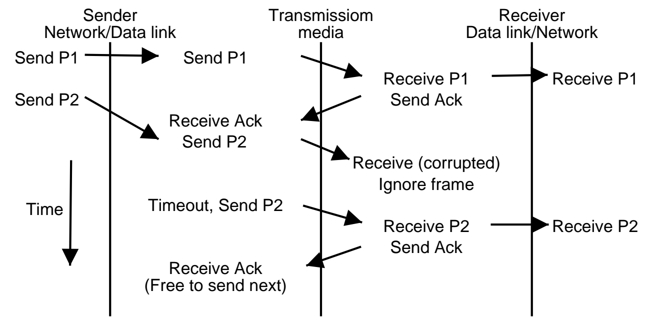
Stop and wait

- It is possible to **leave the problem there**. Upper layer can ask for resending once it finds that the data it expects is not there.
  Indeed, upper layer has to do it anyway: network layer drops packets.
- But this is **not very efficient**: communication will be much faster if the sender don't need to go back resending every now and then.
- One mechanism to deal with it: **acknowledgement**:
  - Sender: after each frame is sent, it will **stop**.
  - Receiver: after getting a frame, it sends back an **acknowledgement**.
  - Upon receiving such an ack, sender will **send the next frame**.
- We call this **stop-and-wait**, describing what is done by the sender.
- This **ignores the possibility of error**, so it is in fact unusable.

Timer: dealing with errors

- How the problem show up? The sender would wait for an acknowledgement that **never show up**.
- A simple idea: **give the sender a timer**, which is started whenever a frame is sent.
- The timer would fire at a time **after** the frame would normally get acknowledged (i.e., 1 round-trip delay, plus processing time).
  This is possible because we assume nearly constant delay.
- If an acknoweledgement is received, the **timer is cancelled**, and the sender sends another frame and wait again.
- Otherwise, the timer would fires at some time. Then **the sender send the old data again**.
- The receiver would simply **process all frames** and send verified data to the network layer and send the acknowledgement accordingly.

Actual events



What if, after data receives at the receiver, the **acknowledgement gets corrupted**?
Corruption can occur for all frames, not just for data frames.

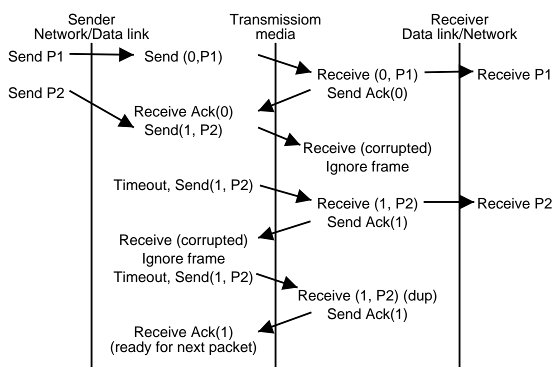
Duplicated frames and sequence numbers

- The sender will timeout, sends again… and the receiver will think it is a new frame and **send it to network layer again**: duplicated frame.
- Is this bad? We can **leave it to upper layer** to deal with it.
  And again, transport layer has to do that anyway, since network layer creates duplicates as well.
- This is not **efficient**: the network layer would have a lot of duplicated frames to handle, so it becomes more **congested**.
- Actually, handling duplicates in transport layer is more difficult, so it's better to deal with it when it is under control.
- How to prevent it? Simple: add a **sequence number**, so that the receiver can distinguish a new frame and an old one.

Positive Acknowledgement with Retranmission (PAR)

- **Sender** keeps a sequence number, the next frame to sent. **Receiver** keeps a sequence number, the last frame received. They are 0 at start.
- Whenever a new packet is sent, the sequence number is incremented. But a new packet can be sent **only if the sender received the acknowledgement**, just as before.
- **Receiver**: a duplicate show up as an **old sequence number**. In this case, the acknowledgement is sent again. Otherwise, it increments the sequence number, send ack, and pass the data to network layer.
- In practice, acknowledgement has sequence number as well, to cater for the possibility that the datalink layer can acknowledge too late.
  If an ack comes too late, we will get a duplicated acknowledgement.
- For stop-and-wait, **1 bit sequence number** is enough.
  Once the receiver receives a data with sequence number 1, there can be no data with sequence number 0 left in the network, due to non-reordering assumption.

Actual events



This will actually work, so its time for performance tuning.

Idea of piggybacking

- Now acknowledgement contains 1-bit data: the sequence number.
- But other than that, we also has to send a **checksum**, and then a **code telling that it is an acknowledgement**, and all the framing codes.
  So a frame contains a header, together with a trailer which is the checksum.
- So the frame to send is actually not 1 bit, but instead a dozen bytes.
- Okay if frames are typically long, but wasteful if otherwise. (The overhead is too large to send a short frame.)
- A simple idea: most channels are **duplex**, i.e., have traffic in both directions. If the reverse channel will send a frame soon, the ack can be put there for a free ride: "**piggybacking**".
- What is "will send a frame soon"? The receiver will have a timer, started on frame arrival. If a frame is send before the timer fires ,the acknowledgement is piggybacked.

Actually improving performance

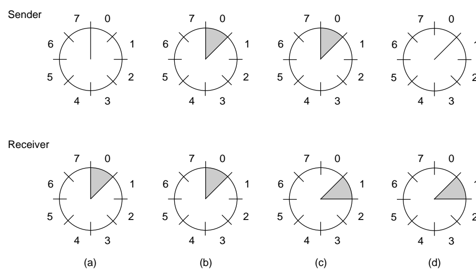
- Difficulty: used this way, piggybacking **reduces** performance: ack is sent late, so next frame will be sent late, leaving an **idle channel**!
- In fact, the channel utilization before piggybacking is not good either: most of the time it is idle waiting for acknowledgement.
- When is the problem introduced? When we require that no frame will be sent until an acknowledgement is received.
- Can we **continue to send new frames** without the acknowledgement?
- If an error occur, we have to "resend" the old frames. So **data link must store some old frames** ready for resend.
  So we must increase the buffer size of data link.
- Another question… Currently we use a 1-bit sequence number to distinguish new and old frames. How many bits we need if we can have many outstanding frames?

Sliding Window

The notion of **sending and receiving windows** gives the proper terminlogy for us to talk about these problems.

- A window is a **consecutive set of sequence numbers**. E.g., for a 3-bit sequence number, it might be 0–2, 3–7, or 6–2 (meaning 6, 7, 0, 1, 2).
- The sender keeps a **sending window**, which is from the first **unacknowledged** frame it sent to the last frame it sent.
- The receiver keeps a **receiving window**, which contains the set of sequence numbers of frames that it can handle currently. The first number should be the sequence number of the next one to forward to network layer.
- Stop-and-wait correspond to the sender having a maximum window size of 1, where receiver always has a window size of 1.

Example: stop and wait



At the beginning, sender window is of size 0. When it sends, the size grows to 1, until it is acknowledged.

The receiver initially has window at 0, and after accepting one frame, the window moves to position 1 (note that it never grow or shrink).

What to do if…

- Sender **sends a frame**: a timer is for the frame, and the right end of the window is advanced. If the maximum sender window size is reached, stop getting packets from network layer until window size shrinks.
- Sender **receives** a (perhaps piggybacked) **acknowledgement**: if it is within the sender window, advance left end of the window to it. All corresponding timers are cancelled.
- Sender **timeouts**: resend all unacknowledged frames in window.
- Receiver **receives a frame** (without error): ???? (forward to network layer, advance receiving window). Setup timer for piggybacking.
  What to do in ???? ?
- Receiver **sends a frame**: the sequence number of the last frame is piggybacked, and the timer is cancelled.
  Receiver only has 1 timer, while sender has 1 for each unacknowledged frame.
- Receiver **times out**: generate a frame for the acknowledgement.

How to forward to network layer

When a frame is received…

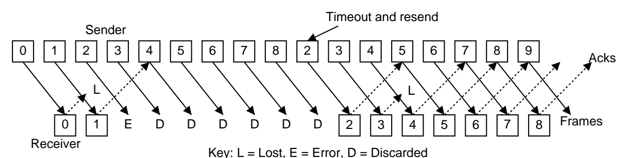
- If the frame received is **not in receiving window**, simply discard it.
- Otherwise, if the frame received is **not the first in receiving window**, store it in buffer, but don't give it to network layer yet. The receiving window is not moved either.
- Otherwise, the frame received is the **first in receiving window**. Forward it to network layer. Do the same for all stored packets immediately after that, until we find a sequence number that we don't have the frame. The receiving window is advanced to that number.

Degenerated case where the **receiving window size is 1**: the middle case never occur, and each time we always forward 1 frame to network layer.

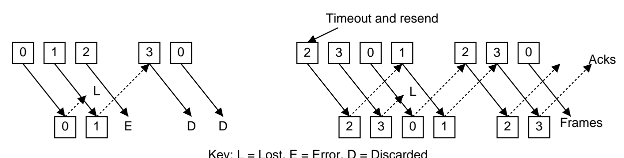
We call this degenerated case "go-back-n", named after what happens to the sender when it cannot receive a frame.

Examples: Go back n

Number of sequence numbers = 16, max sending window = 15:



Same, but number of sequence numbers = 4, max sending window = 3:



Window size

When the number of sequence numbers are reused, is it possible that the it is mis-interpreted?

- If the maximum sender window size is SW, the sender has at most that number of outstanding frames, say k to $k + SW - 1$.
- The receiver might think that it has received all the frames, so the receiver window can advance to at most $k + SW$ to $k + SW + RW - 1$, RW being the size of receiver window.
- **All these sequence numbers must be different.** Otherwise, k is actually in the range between $k + SW$ to $k + SW + RW$. If all acknowledgements are lost, and the sender resend frame k, then the receiver will mistaken that it is a new packet.

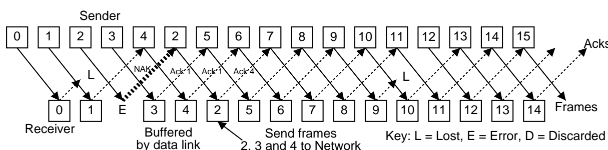
In other words, $SW + RW$ **cannot be larger than the number of available sequence numbers** to guarantee that this protocol works.

Error and improvement

- Go-back-n has the behaviour that **whenever an error occurs, all frames following it until will be resent**.
- It improves when we **allow a larger receiving window**. When the resent frame is finally acknowledged, new frames can be sent again.
- But the time between the resent and the acknowledgement of it are still wasted. (Try it!)
- The problem: **acknowledgement arrives too late** to restart a stalled communication channel.
- We can further improve it if **the receiver can somehow tell the sender early** that a frame is lost: **Negative Acknowledgement** (NAK).
- How to know that? If a corrupted frame is received, or if a sequence number is skipped, then a frame is known to be missed.

Example: selective repeat

Number of sequence numbers = 16, max sending window = 8, receiving window size = 8:



Due to the NAK, frame 2 is resent much earlier then when it eventually timeouts, and the problem is corrected by resending only frame 2. The channel is kept completely busy.

One fine point: one should be careful not to resend NAKs until it times out, otherwise it can happen that NAK is sent a few times and bandwidth is wasted on unnecessary resending.

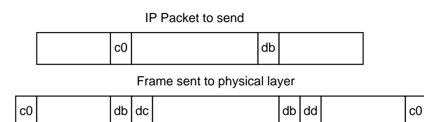
Datalink in practice

Internet datalink

- The Internet TCP/IP protocol stack does **not** mandate a particular datalink layer protocol.
- TCP/IP puts datalink into the lowest layer, “host-to-host access”. It allows **any** implementation, as long as **IP packets can be sent** over it.
- In practice, a few protocol are in frequent use:
  - **Dialup links** and **connections between routers**: uses PPP (Point-to-point Protocol) (RFC 1661–1663), or for older systems, SLIP (Serial Line IP, non-standard).
  - **Ethernet**: uses its own Ethernet protocol encapsulation (RFC 894 and 1042).

SLIP

SLIP is one of the simplest datalink protocol that actually in use today. It simply does one thing: framing.



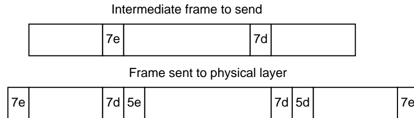
So it **doesn’t provide any error control or flow control measures**. It is byte-oriented, i.e., perform byte stuffing rather than bit stuffing.

Its primary use is in slow (but quite reliable) serial links that is popular in the early days of Internet, which explains its lack of features.

If computers are faster than communication, there is little point for flow control. At that time, 19200bps (i.e., 1920 bytes per second: apart from the 8 bits of data, 2 more bits are needed to identify start and end of bytes) is considered to be extremely fast.

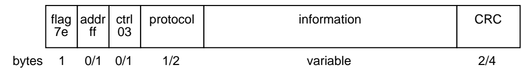
Framing in PPP

- There are two modes of PPP framing. Usually, **fixed** serial lines uses **bit stuffing** we have described earlier.
- For dialup connection using modems, bit stuffing is not a good idea, as the hardware of modem transfer bytes, not bits.
- The FLAG character is 7e, the ESC character is 7d. The character after ESC is the original character with 6-th bit inversed, so most characters can be escaped.
  This is important: many modems interprets control characters, which would cause unexpected failure of the link.



PPP Frame formats

- Unlike SLIP, PPP frame after unstuffing is not the network packet:



- There are quite a few "constant" fields . The are added so that they look like a "High-Level Datalink Control protocol" (HDLC) frame.
- Protocol field allows the link to be used for multiple types of network packets. So PPP can be used for **many protocols at the same time**.
  We say the frames are "self-identifying". IP data are sent with protocol = 0x0021.
- CRC allows error detection, while flow control is not done by default.

LCP and NCPs

- One special type of frame (with protocol = 0xc021) is used for "Link Control Protocol" (LCP) packets, used to **negotiate link parameters**.
- Link parameters include things like omission of constant fields ,the use of the selective repeat flow control protocol, etc.
- It is also used to shutdown an unneeded link.
- Each network protocol type can define its own **network control protocol** (NCP). E.g., IP has its NCP with protocol = 0x8021, to communicate IP addresses for the two sides.
- The important point:
  - A layer can start in a default, "safe" state at the beginning, and be **configured subsequently** with the upper layer protocols (e.g., LCP).
  - Another way to do configuration of a layer (e.g., IP) is to have a **separate, simpler protocol** (e.g., NCP) for configuration.

Ethernet framing

The Ethernet (RFC 894) uses a very different strategy as for framing.

- As we already know, Ethernet uses **Manchester encoding**. When a frame is sent on Ethernet, the voltage in the link will **alternate between 0 and 1 repeatedly**.
- On the other hand, on an **idle** link, the voltage will stay at high level. So it naturally provide framing, in physical level.
- The beginning of any Ethernet frame start with a **preamble**, a pattern containing 64 bits of alternating 1 and 0. This allows the receivers to **synchronize** with the sender.

As we already mentioned, no stuffing is needed due to the generosity of the physical layer.
  Which in turn is due to its broadcast nature. See next lesson.

Ethernet frame format

Ethernet frames share many characteristics with PPP frames.

	Preamble	Dest addr	Source addr	Type	Network Packet	Pad	CRC
Bytes	8	6	6	2	0-1500	0-46	4

- CRC is common, Type serves the same purpose as "protocol" in PPP.
- But LCP and NCP are not used in Ethernet. There is nothing to configure in link layer. For network configuration,ARP and RARP are used.
  We will study these protocol later.
- Source and destination addresses are 48-bit hardware addresses of Ethernet interface ("LAN cards").
  Ethernet is a broadcast medium, so such addresses are needed.
- Other than header and CRC, Ethernet frames must be of size 46–1500. If network packet is too short, it will be padded.
  Why a minimum? We will see it in next lesson.