The Datalink Layer

COMPUTER NETWORKS A.A. 24/25

Leonardo Maccari, DAIS: Ca' Foscari University of Venice,
leonardo.maccari@unive.it

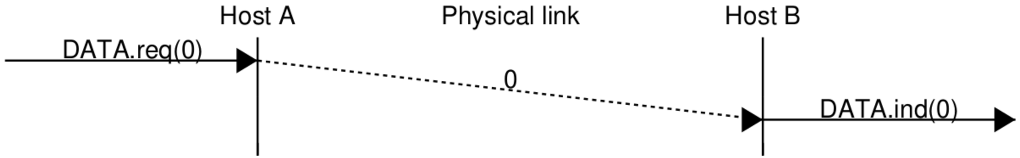
Venice, fall 2024

License

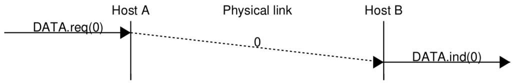
Sect. 1 Notation

Time-sequence diagrams (TSD)

- A time-sequence diagram describes the interactions between communicating hosts.



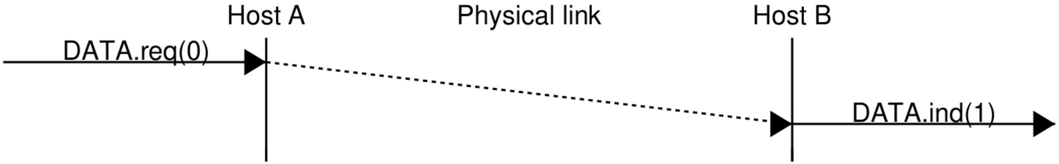
Time-sequence diagrams: features



- time goes from up to down.
- We describe sending a bit as a high-level system call `DATA.req(bit-value)`
- We describe receiving a bit as a high-level system call `DATA.ind(bit-value)`
- Transmission is not instantaneous

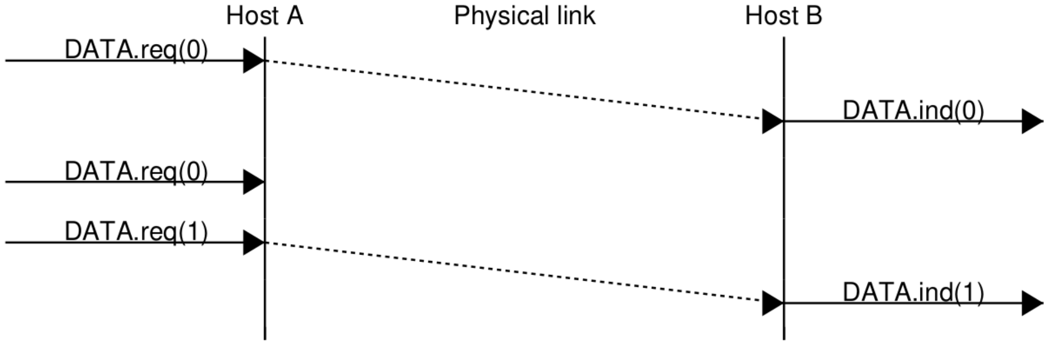
Communication errors in TSD: noise and distortion

- Bits can be *flipped*, that is, noise makes the receiver think the bit is different than what it was when it was sent.



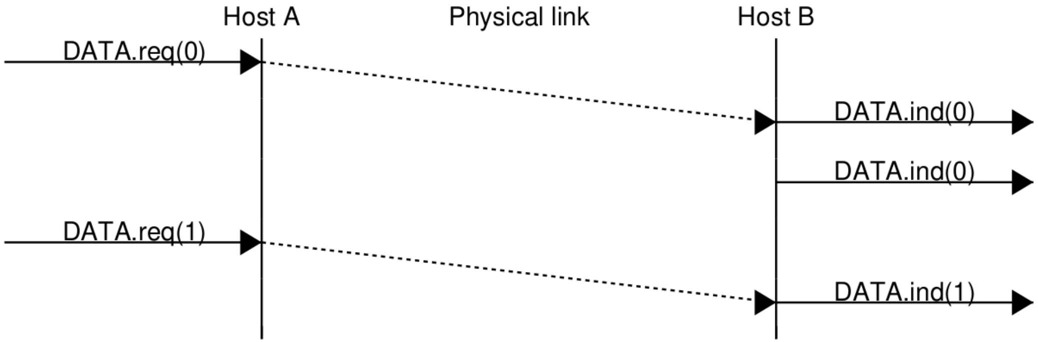
Communication errors in TSD: de-synchronization

- The receiver may *sense* a lower number of bits than the transmitted ones, due to misalignment in the clocks



Communication errors in TSD: de-synchronization

- The receiver may *sense* a higher number of bits than the transmitted ones, due to misalignment in the clocks



Mancherster Encoding

- Keeping clocks aligned is a hard task.
- If the sender sends a long train of signals that never change (i.e. a long sequence of 1 values), the receiver clock may get misaligned
- Instead of using the intuitive encoding (low $\rightarrow$ 0; high $\rightarrow$ 1) it is more convenient to force the communication media to always flip from low to high at every symbol.
- If at every period the level of the signal changes, the receiver can resynchronize the clock with the sender.

Sect. 2 The Datalink Layer

Framing

- Computer scientists are usually not interested in exchanging single bits between two hosts
- They prefer to write software that deals with larger blocks of data in order to transmit messages or complete files.
- The basic unit of information exchanged between two directly connected hosts is often called a *frame*.

Framing (2)

Frame: A sequence of bits that has a maximum length and a particular syntax or structure

How does the receiver know the beginning and the end of a frame?

Simple solution

- Ideally, the physical media simply stops sending data when the frame is over
- However, we have seen that modulating a signal means that there is a carrier signal.
- In some cases the carrier is always on, so we we can't simply stop sending signals
- Then we need to encode the beginning and end of a frame.
- For instance let's say that the sequence '01111110' is used to mark the beginning and the end of a frame, a *frame boundary marker*.

Bit Stuffing

- problem solved? not really, what happens if the sender wants to sent exactly that sequence inside the frame?
- The receiver will interpret it as the end of the frame and this is an error
- The sender must ensure that there will never be six consecutive 1 symbols transmitted by the physical layer inside a frame.

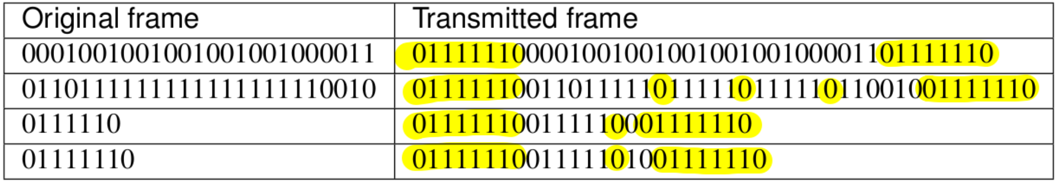
Bit Stuffing: encoding

- First, the sender transmits the marker, i.e. 01111110.
- Then, it sends all the bits of the frame
- It inserts an additional 0 bit after each sequence of 11111 (5 ones)
- This ensures that the data frame never contains the marker.
- The marker is also sent to mark the end of the frame.

Bit Stuffing: decoding

- The receiver performs the opposite to decode a received frame.
- It first detects the beginning of the frame thanks to the 01111110 marker.
- Then, it processes the received bits and counts the number of consecutive bits set to 1.
- If a 0 follows five consecutive bits set to 1, the next bit is removed. It must be a zero inserted by the sender.
- If the marker is detected the frame is over.

Bit Stuffing: examples



Original frame	Transmitted frame
0001001001001001001000011	01111110000100100100100100100001101111110
0110111111111111111110010	011111100110111110111110111110110010 01111110
0111110	011111100111110001111110
01111110	0111111001111101001111110

Protocol Overhead

Protocol Overhead: Extra data that must be transmitted to make the communication possible. The overhead reduces the available bit-rate given by the Shannon and Nyquist formula.

Bit Stuffing: Overhead, worst case

- Bit stuffing increases the number of bits required to transmit each frame.
- First it adds a constant overhead of 2 bytes (the markers)
- Then, assume data is made of all 1 bits. Every 5 bits you need to add a zero bit.
- This added bit is not transporting any information, it will be removed at destination
- You are wasting 1/6 of the bit-rate.
- It means that if your link can provide 6Mb/s, you use only 5

Bit Stuffing: Overhead, average case

- This is the worst case, in average, you will waste something between 0 and 1 Mb/s.
- To compute the average case we need to make some assumption on the probability of having zero and one.
- Let's assume zero/one is a Bernulli variable with $p = 0.5$
- You add a zero when you have 5 consecutive ones, that is $\frac{1}{2^5} = \frac{1}{32}$ of the cases.
- That means that on average $\sim 3\%$ of the capacity is wasted (more on this in the next exercise)
- In the real world, you just don't know the average value, because you can not forecast how many bits will be zero or one.

Exercise

- Assume you always transmit 64 bit frames
- Assume your frame marker is `011110`
- What is the average overhead of your link layer?

Exercise

- First of all, every 64 bits, you add $6 \times 2 = 12$ bits for the markers

Exercise

- First of all, every 64 bits, you add $6 \times 2 = 12$ bits for the markers
- You must avoid to have 4 ones in a row, so every third *one*, you need to stuff a *zero*, that happens with probability $\frac{1}{8}$ given a 3-bit sequence

Exercise

- First of all, every 64 bits, you add $6 \times 2 = 12$ bits for the markers
- You must avoid to have 4 ones in a row, so every third *one*, you need to stuff a *zero*, that happens with probability $\frac{1}{8}$ given a 3-bit sequence
- How many do we have? Example: we send $11011011100\ldots01011011101010$

Exercise

- First of all, every 64 bits, you add $6 \times 2 = 12$ bits for the markers
- You must avoid to have 4 ones in a row, so every third *one*, you need to stuff a *zero*, that happens with probability $\frac{1}{8}$ given a 3-bit sequence
- How many do we have? Example: we send $11011011100\ldots01011011101010$
- We need to check every possible consecutive subset of 3 bits

$$\underbrace{110}_{1}\,11100010\ldots11011101010$$

$$1\,\underbrace{101}_{2}\,1100010\ldots11011101010$$

$$\vdots$$

$$11011011100\ldots01011011101\quad\underbrace{010}_{64-(3-1)}$$

Exercise

- So in the end, you will have an average of $\frac{62}{8} \sim 8$ bit overhead[1]
- plus the markers, it makes 20 bits to send 64 bits of useful data: $\frac{20}{84} \sim 23\%$ of the capacity is lost.

!

Note: the larger the marker, the more fixed overhead you have, but the smaller the probability of stuffing bites. So a good rule is: use large markers, but try to have big frames to compensate for the fixed overhead given by the markers.

[1]Note this is not strictly correct because after you stuffed a zero, the probability you have three ones is zero deterministically for the next 3 subsets.

Exercise (2)

- We use a link with the following parameters and the marker of length 6:
  - B = 20MHz
  - M = 16
  - $S/N = 222$
- What is the probability that the whole frame is missed?
- In order to miss a whole frame, the receiver needs to misdetect the frame markers, failing to detect the beginning or the end of a frame.
- This is the worst case, because the radios become de-synchronized and all data is lost.

Bit error rate: BER

BER: The probability that a single bit is wrongly decoded

- Shannon law tells us that $C_S = Blog_2(1+222) \simeq 156Mb/s$
- Nyquist says that we are transmitting $C_N = 2 \times 20log_2(16) = 160Mb/s > C_S$
- This means that in average 4 Mb/s contain errors: $BER = \frac{4}{160} = 0.025$
- If only one of the marker bit is wrong, the marker is not detected, the frame is lost. Markers are 6 bit longs but we have two.
- The probability of correctly receiving 12 bits is $p = (1-0.025)^{12} \simeq 0.74$
- with a BER as small as 2.5%, 26% of the frames will be lost/corrupted. A lot! a longer marker is needed.

Frame Preamble

- In modern networks, bit stuffing is not used. It is replaced by a long preamble.
- The preamble is used by the receiver to detect the beginning of the frame, it admits errors, but is it long enough so it is impossible to detect it by mistake
- Once the preamble is detected the frame starts, and the frame header contains the length of the frame, no end marker is needed.
- If the end marker is absent, no bit-stuffing is needed.

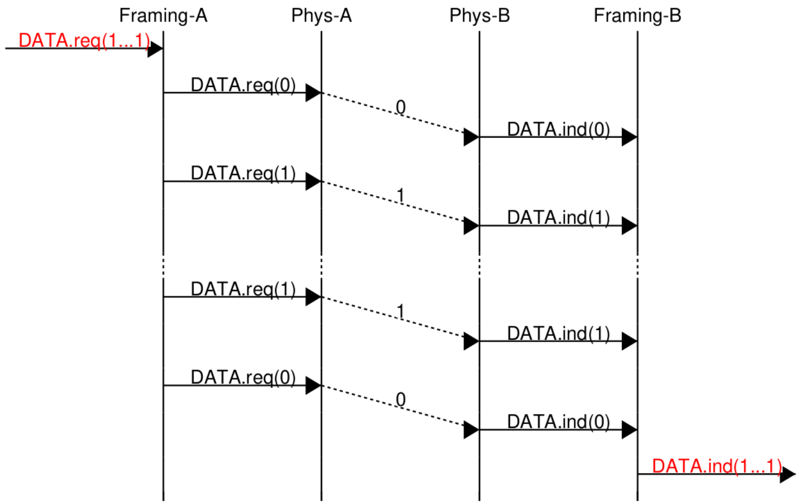
Protocol Overhead

- Bit stuffing is only one of the sources of overhead
- Note that bit stuffing nowadays has been superseded by other techniques, but we keep referencing to it because it is a good example of Datlink layer technique.
- However, overhead is introduced by many other techniques, the remediation due to a lossy link, or by the fact we want to transport meta-data to make the communication reliable/multiplexed etc. . .

Character Stuffing, Octet Stuffing. . .

- The same concept can be applied to other sizes of the data
- For instance, instead of stuffing one bit, we could use a longer marker (a sequence of chars) and stuff a whole byte (a char) when needed
- The basic concepts remains the same.

Frames Exchange



Framing and Layers

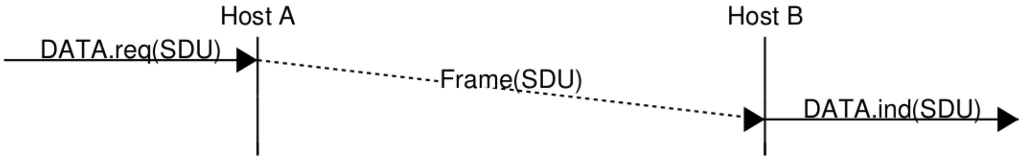
- In the previous image you have a good example of layering in networks
- We already own a physical layer that supports the `DATA.req()` primitive
- We add a layer that uses a similar primitive to send frames (it takes more than one bit as a parameter)
- The Framing layer will also take care of adding the markers and the stuffed bits, and the higher layer will never know.
- This might look trivial, but it is fundamental. If you are a programmer of the outer layer, you don't need to know the inner layer, just the interface to it.

The Datalink Layer

↳2.1 Acknowledging Frames

Service Data Units

- The datalink layer receives what are called Service Data Unit, SDUs.
- It will then transform them into frames as we have seen before
- We have the following



- let's now assume that the communication is perfect, there are no errors at the link layer

Example: Video Call

- `DATA.req(SDU)` is a call from a piece of software.
- This is an application running on host A that decides that it will send data.
- For instance, host A is making a video, and each SDU is a new video chunk, let's say one SDU contains one second of the video
- The video application calls `DATA.req(SDU)` and pushes the data to the lower layer

Software calls (2)

- The application on the receiver needs to play the video.
- Every second it will check if there is a new piece of the video and call `DATA.ind(SDU)`

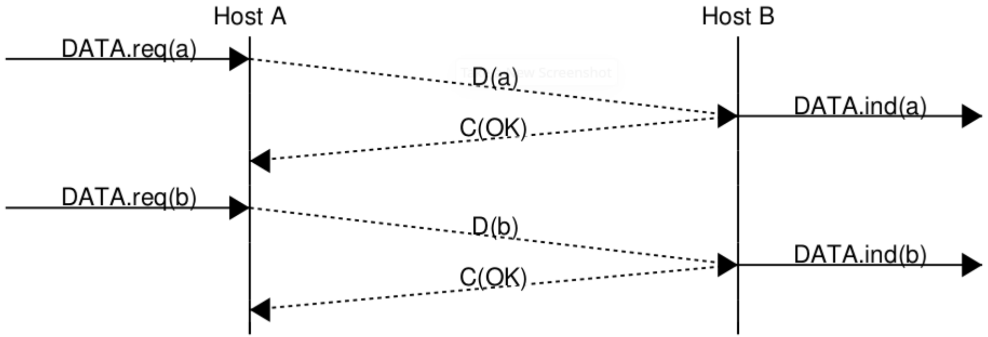
The case of a Slow Receiver

- What if the player at the receiver is too slow to process the video?
- Maybe it takes 1.1 second to process the video chunk and show it to the user
- Then Host B should buffer the video chunks in a buffer.
- When the buffer gets filled up, it will start dropping video chunks.
- In this situation it makes sense if the receiver can communicate to sender that it should slow down the generation of new frames.
- This could be done with **acknowledgment** frames.

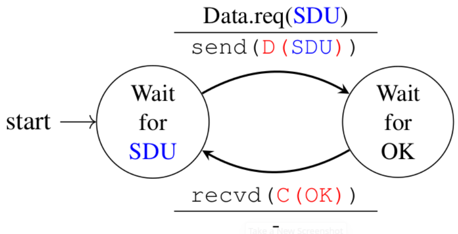
Acks

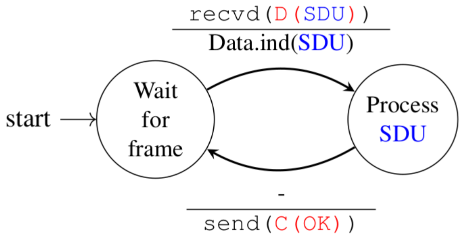
- An acknowledgment is a packet that does not carry any data, it just signals that the previous data was received correctly.
- There must be a way to distinguish a data packet from an ACK
- We separate the frame in two parts, the first part is a *header*, it does not contain data, it contains one single bit: 0 for a for data, 1 for ACKS
- The second part is the *payload*, that is, the real data to be exchanged
- The whole frame (header + payload) will eventually be bit-stuffed
- We have a new packet exchange

Acks

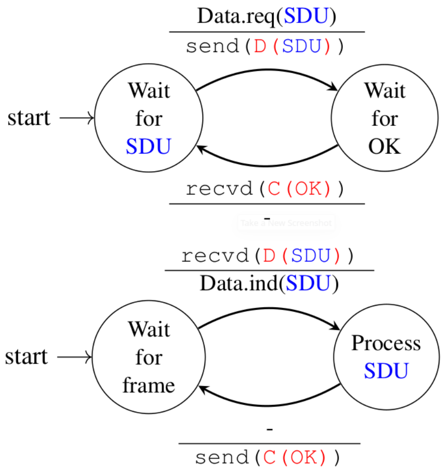

Host A
Host B
DATA.req(a)
D(a)
DATA.ind(a)
C(OK)
DATA.req(b)
D(b)
DATA.ind(b)
C(OK)

The Datalink layer State Machine





The Sender and Receiver State Machine



- The sender (upper half of the image) and the receiver (bottom half) have two states
- On the top of the edge label you have the event that triggers the change of state
- On the bottom of the edge label you have the event that is generated during the change of state
- D stands for Data, C for ACK. If the communication standard uses bit stuffing, the bits will be modified as needed.

Slowing down

- We now have a system that allows the sender to send frames with a mechanism to reduce overflow at the receiver.
- It works as long as the link layer introduces no errors.
- We now introduce errors. . .

The Datalink Layer

↳2.2 Dealing with Errors

Link Error

- A frame may contain errors: flipped bits, missing bits, added bits
- The first problem to be solved is how to detect that a frame is affected by errors

Error Detection

- The simplest **error detection** code is the parity bit, that can be even or odd
- The parity bit is added to the frame to make the number of ones even, or odd
- The receiver then checks the parity and removes the bit

Adding the Parity Bit at the Sender

- Assume the information is 0011011
- Since the number of ones is even, odd parity adds 1, even parity adds 0
  - with odd parity the data become 0011011<u>1</u>
  - with even parity the data become 0011011<u>0</u>
- Computationally, an even parity bit is the sum in modulo 2 of all the bits in the frames

$$(0+0+1+1+0+1+1) \mod {}_2 = (100) \mod {}_2 = 0$$

While the odd parity is the same, plus one.

Checking parity at the Receiver

- Let's assume odd parity is used, then the receiver knows the frame must be made of an odd number of ones.
- If this does not happen, there was a transmission error:
    - 111011011 is received: OK, 7 ones
    - 001100110 is received: NOK, 4 ones
- If there is no error, the receiver just removes the parity bit and passes the SDU to the upper layer
- In case there is an error the receiver will do what is appropriate, that is, ask the sender to send the frame again, or just drop it, depending on the kind of the design of the Datalink layer.

Limits of the Parity Bit

- If there is an even number of errors in the message, the parity bit can not detect them.
- Example with even parity:
  - Original message 00110101
  - Original message + parity: $0011010\underline{1}0$
  - Received message: $\mathit{11}110101\underline{0}$ → Error not detected!
- The receiver will not detect the error.
- Note that the error can be in parity bit itself too:
  - $0\mathit{1}110101\underline{\mathit{1}}$→ Error not detected!

The Internet Checksum

- A parity bit is a kind of checksum, i.e. a sum that is used to make a check on data
- One family of popular checksum is called CRC (cyclic redundancy check)
- The Internet Protocol (IP) that we will study later on uses another checksum, the so-called Internet Checksum
- The Internet checksum is specified in an RFC, a document that specifies Internet standards (more on this in future lessons)
- `https://www.rfc-editor.org/rfc/rfc1071`

Checksum Definition

Internet Checksum[2]: The checksum field is the 16-bit ones' complement of the ones' complement sum of all 16-bit words.

[2] Actually, at the datalink layer, CRC is used. However, CRC is more complicate and we want to focus on upper layers, so I will introduce IP checksum as an example of a checksum. Just recall this is not used in the datalink layer.

Internet Checksum

- ones' complement ($\sim$): the inverse of a binary number

$$\sim 1100 = 0011$$

- ones' complement sum ($+'$): sum, and if there is a carry, sum the carry to the result. Note, the checksum has a fixed size, you can not enlarge it as needed.
- Consider a 4-bit sum:

$$1100 +' 0010 = 1110 \text{ (no carry)}$$

$$1100 +' 1010 = 0111 \text{ because:}$$

$$1100 + 1010 = 10110 \text{ the first 1 is the carry, so the result is } 1 + 0110 = 0111$$

Chaining carries

- Note that if we need to sum more than two elements, we can make all the binary sums and then sum all the carries
- $1100 +' 1010 +' 1100 = ?$

$$1100 + 1010 + 1100 = 100010$$

$$1100 +' 1010 +' 1100 = 0010 + 10 = 0100$$

- Note however that if there is a carry from the last sum, you will need to sum it again, and again till there is no carry anymore

The Internet Checksum

- *the 16-bit ones' complement of (the ones' complement sum of all 16-bit words)*
- So first we compute the ones' complement sum of all the 16-bit words
- Then we compute the complement.
- Let's make an example, for readability we use words or 4 bits instead of 16.
- We use the data: 1100 0011 1010 1110 1001

Example

First we compute the normal sum:

$$
\begin{array}{r}
1100+ \\
0011+ \\
1010+ \\
1110+ \\
1001 = \\
110000
\end{array}
$$

This becomes $0000 + 11 = 0011$ and then we make the complement. The checksum is: 1100

In the frame Header

- The checksum is added in another header field in the frame.
- So the receiver receives the data and the checksum.

$$\text{dataframe} = 1100\ 1100\ 0011\ 1010\ 1110\ 1001$$

Verifying the Checksum

- The receiver will simply compute the $+'$ sum on all the data (including the checksum).
- If the result is made of all ones, there were no errors.

Verifying the Checksum

$$\begin{aligned}
&1100+\\
&1100+\\
&0011+\\
&1010+\\
&1110+\\
&1001 =\\
11&1100
\end{aligned}$$

This becomes $1100 + 11 = 1111$: Correct!

Checksum Design

- If you think about it, this is straightforward
- The checksum is the one's complement of the sum of all bytes, so if you sum it again, you will have all ones…
- Note that there are combinations of errors that can still produce a valid checksum. The probability of having some errors but still having a correct checksum is bounded by 2^{-C} where C is the size of the checksum (see the RFC for details)

Checksum Performance

- Checksum computation must be very fast, because you have to do it in a router millions times per second. Notice that:
  - You can start computing the checksum before you completely receive the frame. Every 16 bits you receive you can update the sum. You don't need to store the whole packet and then compute the checksum, that would be slower.
  - Because addition is associative, you can sum the data in the order you want. This means that if you have 64 bit registers, you can use 64 bits sum, and then reduce to 16 at the end. This helps speeding up the computation.

Getting Familiar with RFC

- RFC 1071 explains the IP checksum, `https://www.rfc-editor.org/rfc/rfc1071`
- It also provides source code for C language

Some Consequences

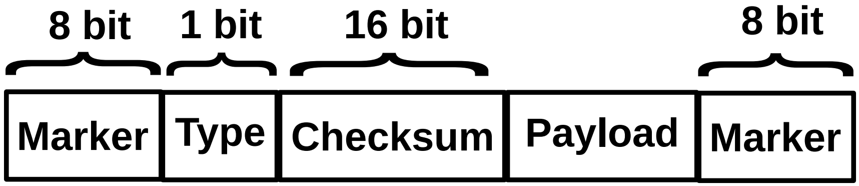
1. The header is now larger: we have 16 extra bits
2. The packet must be byte aligned, that means that the data length must be a multiple of 16 bits
3. If you have less than that, you may need need to *pad*: add some bogus bits at the end
4. The IP checksum is used in several places in the Internet Protocol, we will tell more in future lessons

Exercises

- Assume you received `0100 1010 1111 0110 0011 0101`. This contains the checksum. Can you tell if there were errors?
- Compute the 4-bit checksum `C` of the frame
  `F = 1100 1011 1110 0110 0011`
  then add the checksum to the frame, and verify it is OK.
- Assume the sender sends `[F,C]`, but the receiver receives `[F',C]`, where `F'` is a version of `F` with errors. Can you find `F'` so that the checksum is still valid, even if `F'!=F`?

Conclusions

- We started with a phy layer able to send bits, and we are able to compute its capacity
- We introduced the concept of framing, and the overhead due to the markers and bit stuffing
- We also introduced the frame header, that contains one bit for the kind of frame (Data/ACK)
- We finally introduced the checksum in the header



Overhead Vs Features trade-off

- All the things we introduced add overhead, so they subtract useful resources to the communication
- However, we are providing a better service to the network layer, because now we have:
  - Flow control, that avoids buffer overflow at the receiver (through `ACKs`)
  - Error detection (which we will expand in next lesson)

References

- Cap 2 pag 5-15 (libro Bonaventure)
- Sect 1 of RFC 1071: `https://www.rfc-editor.org/rfc/rfc1071`