

Datalink layer: overview

Lecture 4

Data link

The datalink layer is responsible for framing and error control. In this lecture we look into the simpler case: how to do it when the link is point-to-point.

References:

- CN 3.9 (up to 3.9.6), 5.2, 5.4, 5.5.
- Tlv12.1–2.6.
- RFC 894 (IP in ethernet), 1042 (IP in IEEE 802), 1661 (PPP and LCP), 1662 (PPP framing), 1663 (PPP with ARQ), 1332 (NCP).

Network(0234B)

Network(0234B)-4.1

The datalink layer exchanges **packets** (2-SDU) with the network layer above, and exchange a **bit or byte stream** (2-PDU) with the physical layer below.

1. **Sender side forms a frame from a packet** by adding a **header** with control information, and then adds checksum in **trailer** for error control.
2. Sender adds **framing information** to identify the start and end of the frame, and put it as a bit-stream or byte-stream for physical layer.
3. Receiver monitors the physical layer to find framing information, and **extracts a frame** from the physical layer.
4. Receiver **recomputes the checksum** to check whether there is error.
5. **No error**: receiver extracts and **forwards the packet** to network layer.
6. **Error**: receiver **informs** the sender so that the frame is **resent**.

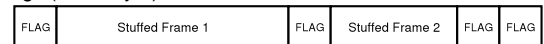
Framing

- What “framing information” would allow the receiver to identify the **start** and **end** of the frame from a **bit or byte stream**?
- Sometimes this is trivial: if the **physical** layer reserves a **symbol** for “**no message**”, we just check where this symbol starts to disappear.
The physical layer does framing for us. E.g., in Ethernet, the mBnB or mBnT coding system leaves symbols for such control info, no datalink framing is needed.
- For physical layers where there is no such symbol, we need to do framing by manipulating the bytes or bits.
- An easy way: 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.
So length is good only if reliable. (Some schemes checksum just the length!)

Network(0234B)-4.2

Byte stuffing

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



- The FLAG byte **never** appears **within** the bit stream. When a frame contains it, the data link convert it to 2 bytes: an **escape** followed by another code. And escape is similarly converted, or **byte-stuffed**.
Just like that we write `\n` and `\\` in a C++ programs.
- There is one restriction, though: this requires the communication channel to be **byte-oriented**. This is actually the case for many **modems**, which is the place where byte-stuffing is most useful.
- Other physical layers are **bit-oriented**, so the data link must find exactly **which bit starts a frame**. A different scheme must be used.

Network(0234B)-4.3

Bit stuffing

- A **special bit pattern** 01111110 is used as FLAG.
This is the value used by PPP.
- For frame data, **add an extra 0-bit after every 5 consecutive 1-bits**. So the FLAG never appear within the frame data.
E.g., if we want to send 0111111100, we instead send 011111011100.
- Upon receiving a sequence of 0111110, the receiver **drops the last 0**.
- Upon receiving a sequence of 0111111, the receiver **declares a frame boundary before the 0**. It then **wait until a 0**, and drop everything before that 0, and start the processing of the a frame.

E.g., `10 111110 10 11111110 00` becomes...

... 1111110 `10 111110 0 10 111110 1110 0 0` 0 111111...

Think about it: will a flag of 1111110 work?

Network(0234B)-4.4

Why error handling in datalink?

In some data link layers, framing is all the things that it does. But in others, error discovery and recovery are performed. Why?

- In general, we need error handling at higher layer, **regardless of whether error handling is also done in lower layer**.
So it is better to add features in higher layers: the “end-to-end” argument.
- But...
 1. Datalink layer is the layer that **knows whether the physical media is noisy or not**. Higher layers don't.
 2. If erratic frames can be stopped in datalink layer, one can **recover sooner** because one don't have to go through the network. One also **reduces the number packets** to send. Both improves performance.

So datalink performs error control if the link is (or maybe) noisy.

Network(0234B)-4.5

Error recovery: assumptions and requirements

So we won't correct errors. If there is an error we need a resend. Let's first see what we expect, and what we want to provide to the network layer.

Assumption:

- Transmission **delay** is nearly **constant** and can be **measured**. This implies that frames are **never re-ordered**.
- If there is an error in the physical link, it is already **detected**. So **every frame received is actually sent** by the sender.
But some frames might be lost altogether.

Requirements:

- **Resend** dropped frames, **don't duplicate** or **reorder** frames, but use only a bounded amount of memory.
- **Transfer speed**: it should utilize physical layer channel well.

Network(0234B)-4.12

ARQ: basic mechanism

- Since frames might get lost altogether, the sender **automatically** resend if the receiver don't tell the sender that the frame is received.
- We call it **Automatic Repeat Request**: repeat request is automatic, the receiver does not have to ask for it. Instead, the receiver send a short frame called the **acknowledgement** to stop the repeat.
- To do this the sender needs two additional pieces of hardware:
 1. A **timer**. A timer is set whenever a frame is sent, which will **time-out** at a time when acknowledgement should have arrived. An acknowledgement would then cancel the timer.
 2. Some **buffers**. When time-out occurs we must resend the old frame. This can be done only if we keep the old frame in some buffer.

Network(0234B)-4.13

The simplest protocol: Stop and wait

Sender:

- Once a frame is sent, store it in the buffer, sets the timer, and **stops**.
- When an ACK comes, cancel the timer, clear the buffer, and **restart**.
Recall that datalink is typically done by hardware, so this restart is done by triggering an interrupt to the CPU.
- When time-out occurs, send again, set the timer, and stop again.

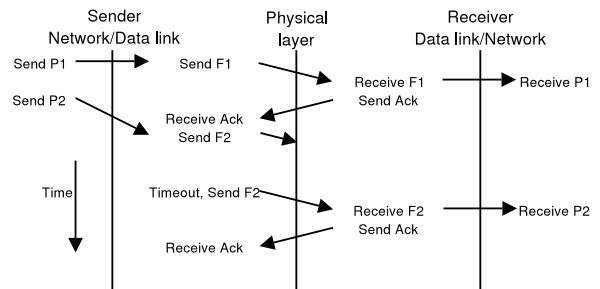
Receiver:

- If a correct frame is received, it sends back an **acknowledgement**, extract the packet, and inform the network layer to get the frame.
Again, by interrupting the CPU.

We call this **stop-and-wait**, describing what is done by the sender once it sends something.

Network(0234B)-4.14

Example events



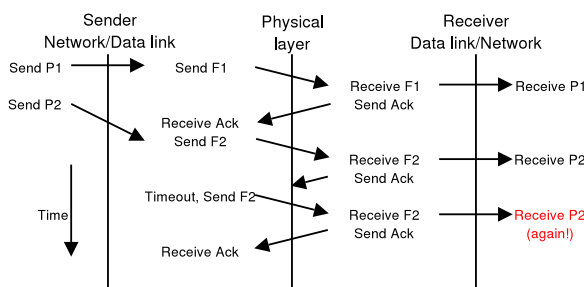
One frame (the first F2) is corrupted, but after time-out it is resent, so the receiver receives exactly one P1 and one P2, in correct order.

Network(0234B)-4.15

Problem 1: Duplicated frames on lost ACK

What if the ACK is lost?

Corruption can occur for all frames, not just for those containing data.

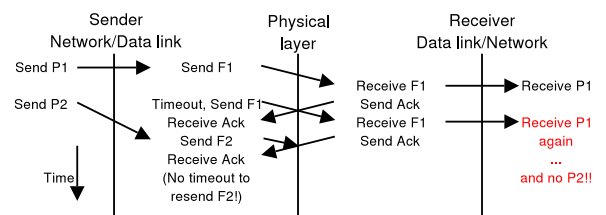


The network layer probably gets a **duplicated packet**.

Network(0234B)-4.16

Problem 2: Duplication/Lost frames on premature time-out

What if the time-out occurs too soon? It's more complicated...



We still get a duplicated packet. But it might also **lost a packet**.

It might be argued that we shouldn't have a wrong time-out in the first place, but even if we are wrong we need to be able to exchange information to correct the problem.

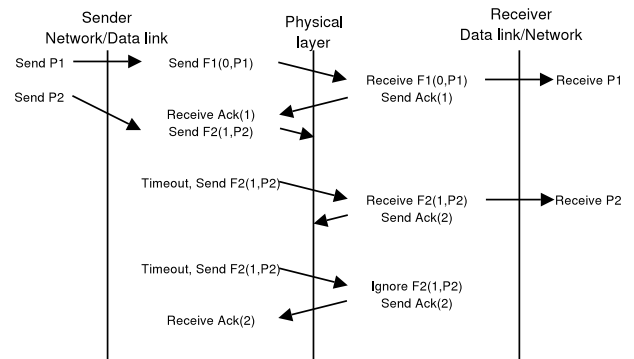
Network(0234B)-4.17

Adding a sequence number

- Both problems can be solved by having a **sequence number** within all frames, whether it stores data or just an ACK.
- The sequence number of frames is **different for different packets**, so allowing old and new ones to be **distinguished**.
- The modification to the protocol:
 - Sender:** Keep a counter S_{recent} initially 0. When a frame is created from the packet, put S_{recent} into the header, and **increment** S_{recent} . Ignore ACKs that does not have the sequence number S_{recent} .
 - Receiver:** Keep a counter R_{next} initially 0. When a frame is received, check its sequence number. If it's R_{next} , **increment** R_{next} and **forward** the packet to network layer. Whether the sequence number is correct, send an ACK of sequence number R_{next} .
So the receiver ACK contains the **next** sequence number it expects.

Network(0234B)-4.18

Example events



Exercise: draw the events with premature time-out.

Network(0234B)-4.19

Improving performance

- If the transmission delay is short, the stop-and-wait scheme is reasonable. But if transmission delay is comparable to the frame length, it wastes a lot of bandwidth.
- Why? The sender **waits** for the ACK, leaving the medium unused.
- E.g., on a 33.6kbps modem connection, sending 1kB of data needs around 238ms. If the ACK arrives only 1s after we send the last bit, the utilization is only 238/1238, i.e., 19%.
- Can we **continue to send new frames** before the ACK comes? Specifically, can we send 6 different frames before getting the first ACK?
- If an error occur, we have to "resend" the old frames. So **data link must store more old frames**, i.e., a larger buffer.
Remember that this is done by hardware, i.e., the network card will be more expensive.

Network(0234B)-4.20

Go-back-n

Let's modify the sender first, without touching the receiver at all.

Sender:

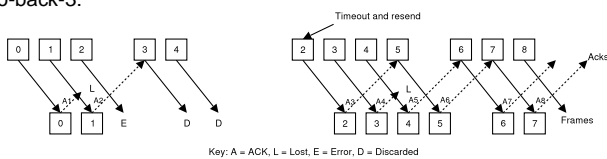
- Once a frame is sent, store it in the buffer, sets its timer. If there are already n frames in buffer (i.e., full), stop.
- When the acknowledgement F comes, cancel all timers before frame F , and remove all frames before F within the buffer. **Allow network layer to send** again if less than n frames are in buffer.
If we receive ACK(F), we assume all frames before frame F are received.
- When time-out occurs, **cancel all timers, resend** everything in the buffer, and set the timer accordingly.

If the buffer is full when a time-out occurs, the sender will **go back n** frames and resend. Hence the name of this scheme.

Network(0234B)-4.21

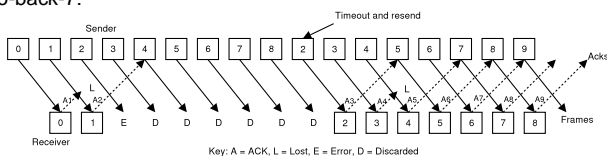
Examples: Go-back-n

Go-back-3:



Key: A = ACK, L = Lost, E = Error, D = Discarded

Go-back-7:



Key: A = ACK, L = Lost, E = Error, D = Discarded

Network(0234B)-4.22

Behaviour on frame error, improvement

- Go-back-n has the behaviour that whenever a **data frame is lost**, all the n frames following it will be **discarded—even if sent correctly!**
- If error rate is low, this is okay. But if **error rate is high**, we would like to reduce the amount of bit-rate wasted for resending.
- The receiving side can **keep correctly received frames** following the omitted frame in a buffer so that the sender don't need to resend them.
- The problem is... we **cannot ACK these frames** (otherwise sender will think the omitted frame is received, too)! So the sender will resend a large portion of the correctly received frames anyway...
- To avoid that, the receiver **asks the sender to resend a frame** if it sees a skipped frame. It is called a **Negative Acknowledgement (NAK)**.
- Hopefully, the receiver sees the NAK, resend correctly, and receive the ACK for it before the real time-out. Then only that frame is resent.

Network(0234B)-4.23

Send and Receive Window

We describe the protocol in terms of **windows** of sequence numbers.

- A window is a **consecutive set of sequence numbers**. We use unbounded sequence numbers, but eventually we only keep a few bits.
- The sender keeps a **send window** $[S_{\text{recent}}, S_{\text{last}} + W_s - 1]$, which are the sequence numbers that it can use for new frames before receiving further ACKs. S_{last} is the last unacknowledged frame (i.e., with least seq no). Its size is at most W_s , the number of buffers of the sender.
- The receiver keeps a **receive window** $[R_{\text{next}}, R_{\text{next}} + W_r - 1]$, which are the frames that it **can** store into buffer. W_r is the number of buffers of the receiver, while R_{next} is the frame it is expecting.

Stop-and-wait: $W_s = W_r = 1$; **Go-back-n:** $W_s = n$; $W_r = 1$.

Stop-and-wait, go-back-n and selective repeat are called "sliding window protocol" because they fit into this framework.

Network(0234B)-4.24

Selective Repeat: sender

Initially, set $S_{\text{last}} = S_{\text{recent}} = 0$, so the send window is $[0, W_s - 1]$.

- When a **frame is sent**, use the sequence number S_{recent} and **increment** S_{recent} . If the send window becomes empty, **stop**.
- When **ACK(N)** comes: if N is not between S_{last} and $S_{\text{recent}} - 1$, ignore it. Otherwise, **cancel all timers** between S_{last} and $N - 1$, **remove** them from buffer, and set $S_{\text{last}} = N$. **Allow network layer to send** again if send window is no longer empty.
- When **NAK(N)** comes: if N is not between S_{last} and S_{recent} ignore it. Otherwise, do all processing for ACK(N), and **resend frame N** within the buffer.
- When timer for sequence N **time-out**, **resend** the corresponding frame in buffer, and set the timer again.
Incrementing $S_{\text{recent}}/S_{\text{last}}$ are called "sliding" the start/end of the send window.

Network(0234B)-4.25

Selective Repeat: receiver

Initially, set $R_{\text{next}} = 0$, so the receive window is $[0, W_r - 1]$. Keep a variable NAK_sent , initially false. When a **frame arrives**, check its sequence number N .

- If N is **outside** the receive window, simply send $\text{ACK}(R_{\text{next}})$.
- If $N = R_{\text{next}}$: set NAK_sent to false. If frames $N, N + 1, \dots, k$ are already received (while frame $k + 1$ is not), **forward all** of them to network layer, and **set** $R_{\text{next}} = k + 1$. **Send ACK**(R_{next}).

This is the only case when we slide the receive window.

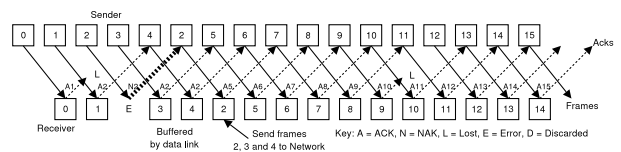
- If N is **within** the receive window, but **not the expected** R_{next} : **Keep** the frame in buffer. If NAK_sent is false, **send NAK** of R_{next} and set NAK_sent to true. Otherwise send $\text{ACK}(R_{\text{next}})$.

We want only one NAK per packet, otherwise the sender will receive a lot of NAK and resend it many times, which is wasteful.

Network(0234B)-4.26

Example: selective repeat

With large enough send and receive window and timeout value, we get the following events when the same frames are lost as our previous example.



The NAK successfully asked **frame 2 to be resent before sender time-out**, so the error is recovered by resending only frame 2. The channel is completely utilized.

Network(0234B)-4.27

Number of bits for the ACK sequence number

There is one remaining problem: we can't keep sequence numbers increasing infinitely unless we have infinitely many bits for it!

Luckily, at any time we only need to distinguish finitely many sequence numbers, so we **only need to send the last few bits**.

- Suppose the last ACK received by sender is $\text{ACK}(L)$. Then the sender will **no longer receive** $\text{ACK}(L - 1)$ or before.
- Before the sender receives a larger ACK, it **can only send up to data frame** $L + W_s - 1$. So the receiver can send only up to $\text{ACK}(L + W_s)$.
- In other words, if the last ACK received is $\text{ACK}(L)$, the sender can only receive $\text{ACK}(L)$ to $\text{ACK}(L + W_s)$.
- So the sender knows which $W_s + 1$ **ACKs are possible** if it remembers L . Hence $\lceil \log_2(W_s + 1) \rceil$ bits are needed to distinguish among them.

Network(0234B)-4.28

Number of bits for the frame sequence number

How about data frames?

- Suppose in the receiver, $R_{\text{next}} = N$. To be correct, the receiver only need to **distinguish data frame** $N, \dots, N + W_r - 1$ **from old frames**.
It is easy to distinguish them from frames that are too new: the newest frame can only be $N + W_s - 1$, requiring just W_s sequence numbers.
- For the receiver to have $R_{\text{next}} = N$, the receiver must have **received the data frame** $N - 1$, sent by the sender.
- So the sender **must have received at least** $\text{ACK}(N - W_s)$. Future data frames have sequence number $\geq N - W_s$.
- So we only need to distinguish among $N - W_s, \dots, N + W_r - 1$. This requires $W_s + W_r$ sequence numbers, and $\lceil \log_2(W_s + W_r) \rceil$ bits.

So stop-and-wait needs 1 bit, go-back-n needs $\lceil \log_2 n \rceil$ bits.

Network(0234B)-4.29

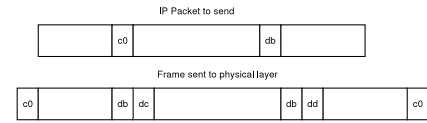
Piggybacking

- ACKs and NAKs need just a sequence number and the indication “ACK” or “NAK” with the peer. But it **also have all the other overheads** of a regular frame: framing, checksum, and the rest of the header.
- This is somewhat wasteful, so many ARQ protocols allow ACKs and NAKs to be **piggybacked** on a regular frame of the **reverse traffic**, i.e., the send-connection of the receiver.
- To send an ACK or NAK with a piggybacking system:
 1. If the reverse connection is **not busy, send immediately**.
 2. Otherwise, **queue** the ACK or NAK in the reverse connection.
 3. Once reverse connection completes the current frame, it checks whether **a new packet can be sent**. If so, **piggyback** the ACK or NAK. Otherwise send the ACK or NAK in its own frame.
Alternatively, a small “ACK timer” may be used.

Network(0234B)-4.30

Datalink in practice: SLIP

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



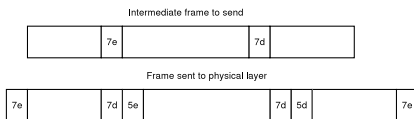
So it **doesn't provide any error control measures**. It is byte-oriented, i.e., perform byte stuffing rather than bit stuffing.

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.

Network(0234B)-4.31

Datalink in practice: PPP (Framing)

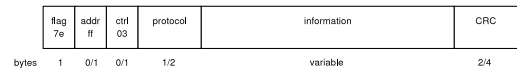
- There are two modes of PPP framing.
- For **fixed serial lines, bit stuffing** is used. For **modem connections** which communicate with bytes, byte stuffing is used instead.
- The FLAG character is 7e, the ESC character is 7d. The character after ESC is the original character with 6-th bit inverted, so most characters can be escaped.
This is important: many modems interprets control characters, which would cause unexpected failure of the link.



Network(0234B)-4.32

Datalink in practice: PPP (Frame content)

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



- There are quite a few “constant” fields . They 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. By default, frames with error are simply dropped with no further recovery. Options can be negotiated to perform go-back-n or selective repeat.

Network(0234B)-4.33

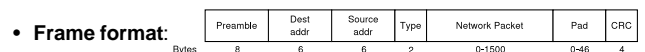
Datalink in practice: PPP (LCP and NCPs)

- 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 ARQ 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.

Network(0234B)-4.34

Datalink in practice: Ethernet

- There are many different versions of Ethernet physical layers, but all reserve symbols for framing. So **datalink doesn't perform framing**.



- **Frame format:**
- To let the sender and receiver clocks to **synchronize**, Ethernet frames **starts with 64 bits of alternating 0's and 1's** called **preamble**.
- As in SLIP, **errors are detected** using CRC, bad frames are dropped, but **no recovery** is attempted.
- As in PPP, there is a **type** to indicate network packet type.
- Ethernet datalink layer performs some routing functions, so it needs a **hardware address** fields . The **pad** field is used for broadcast physical layers. More about these in the next lesson.

Network(0234B)-4.35