Data Link Layer

The **DLL** is responsible for controlling the flow of data in a single link. Unlike the PL which moves individual symbols (bits ot codewords), the DLL controls the movement of a **frame** as a single unit.

The DLL and the PL typically reside on the same card making the boundary between the two a bit fuzzy. Functionally, one may delineate the two layers by assuming that the role of the PL is:

Transmitter: takes one dataword from the output buffer and inserts it into the link.

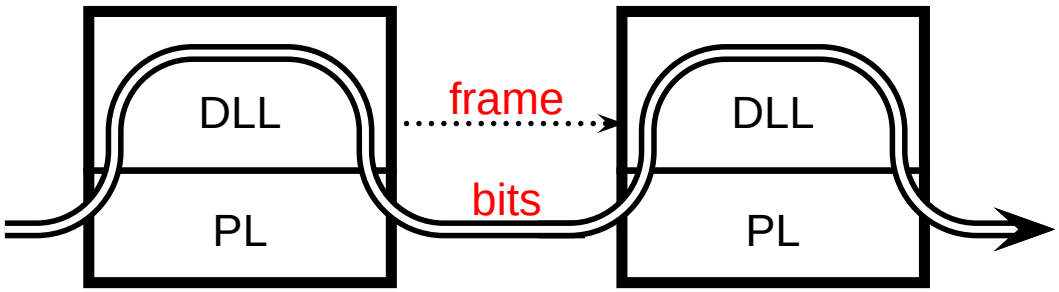
Receiver: accepts one codeword, converts it back into a dataword and places the dataword in the input buffer.

Some error correction and recovery is also done there.

With this description of the role of the PL, the role of the DLL is:

Transmitter: takes a datagram, converts it into one or more frames and places a frame in the output buffer (one framse at a time).

Receiver: takes the contents of the input buffer and attempts to interpret it as a valid frame. If successful it reconstructs a datagram and passes it to the NL. Otherwise, it rejects the invalid frame and initiates error recovery (**FEC** or **BEC**) if desired.



Data Link Layer

The DLL moves NL datagrams over an individual link. The DLL protocol encapsulates a datagram (preceding it with its own header) into a **frame** and passes the frame to the Physical Layer which sends them as independent symbols (e.g. bits) through the physical medium.

There are two types of DLL links:

- A point–to–point link has only one node[a] at each end.
- A shared broadcast link ("multiple access") to which several nodes are attached. These nodes have equal rights to the link, even if their actual role may differ (e.g. router vs. host).

The PL is not aware that a link is a broadcast link. It is a responsibility of the DLL to resolve any problems related to the multi–point nature of a broadcast link.

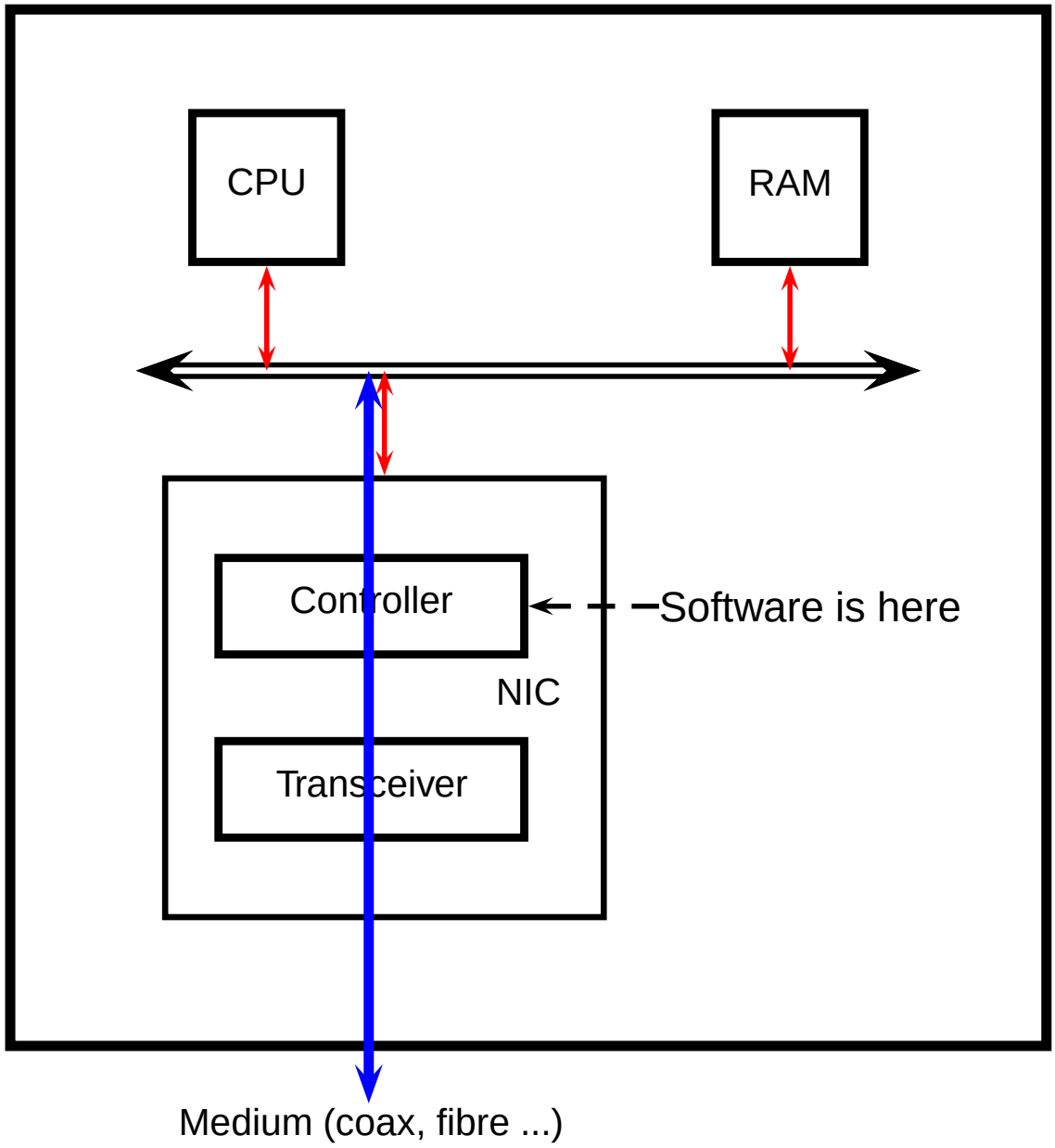
[a] Node = host, router, switch, etc.

MAC protocols

Every link is controlled by a DLL protocol. All these protocols are called **Medium Access Control** protocols. The same protocol must be obeyed by all the nodes on one link, although the same node may use several different MAC protocols if connected to more than one link.

When using a p2p link both nodes connected to it must be compatible but are not always the same, as in 802.11b, 802.11g and 802.11n or in “Fast Ethernet”. Modems and switches usually handle that by trying to use various protocols (mainly transmission rates) until they can recognise the signal coming from the other side.

DLL and PL


CPU
RAM
Controller
Software is here
NIC
Transceiver
Medium (coax, fibre ...)

Framing

Framing is performed at both ends of a link:

- A NL datagram is framed at the transmitter end. It involves encapsulating a datagram (after possibly fragmenting it first).
- An input buffer submitted by the PL is interpreted as a frame at the receiver end. Validation is main task at this stage.

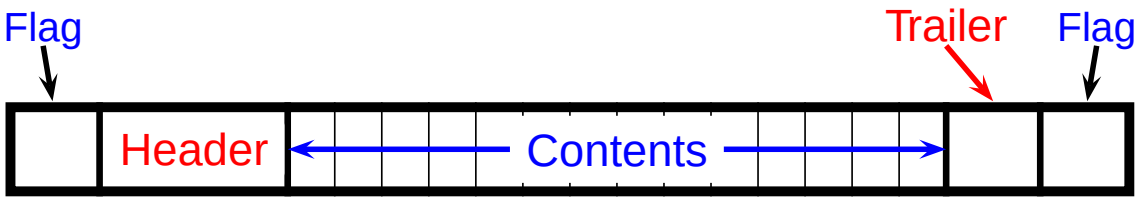
Framing a datagram

The DLL starts by deciding whether to fragment a datagram. If so, it turns it into 2 (or more) datagrams and treats each of them separately (but not independently because enough common information must be forwarded to the receiver to allow the datagram to be reassembled). Fragmentation is required when the DLL suports fixed–size frames only, as in **ATM** or **DQDB** which both require frames to be 53–bytes long.

A datagram is then framed; the **MAC** protocol used may call for byte–oriented or bit–oriented framing protocols.

A frame

there is little difference between byte and bit oriented protocols. All of them require a similar frame structure:



The flags usually are identical sequences of 8 bits (or 1 byte, as desired). Many protocols use 01111110 as a flag, but Ethernet uses 10101011 instead (and no trailer).

Escaping and stuffing

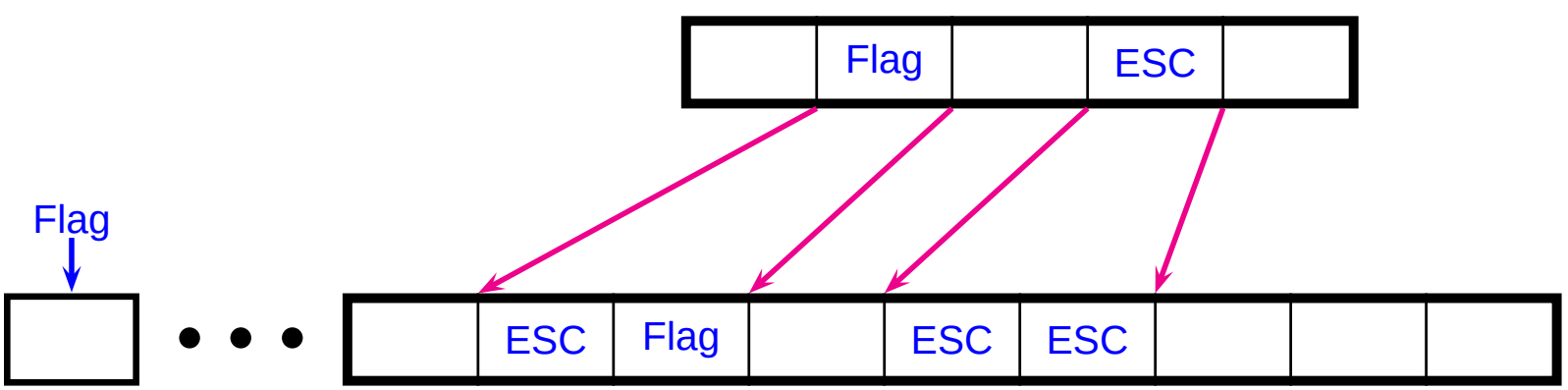
The existence of bit control bit sequences poses an obvious problem: what if an identical pattern appears in the datagram coming from the NL?

Two similar solutions are used:

Escaping used mainly in byte–oriented protocols identifies a specific byte called **ESC** which is placed in front of a data byte identical to a control byte (also in front of a data byte that looks like an **ESC**).

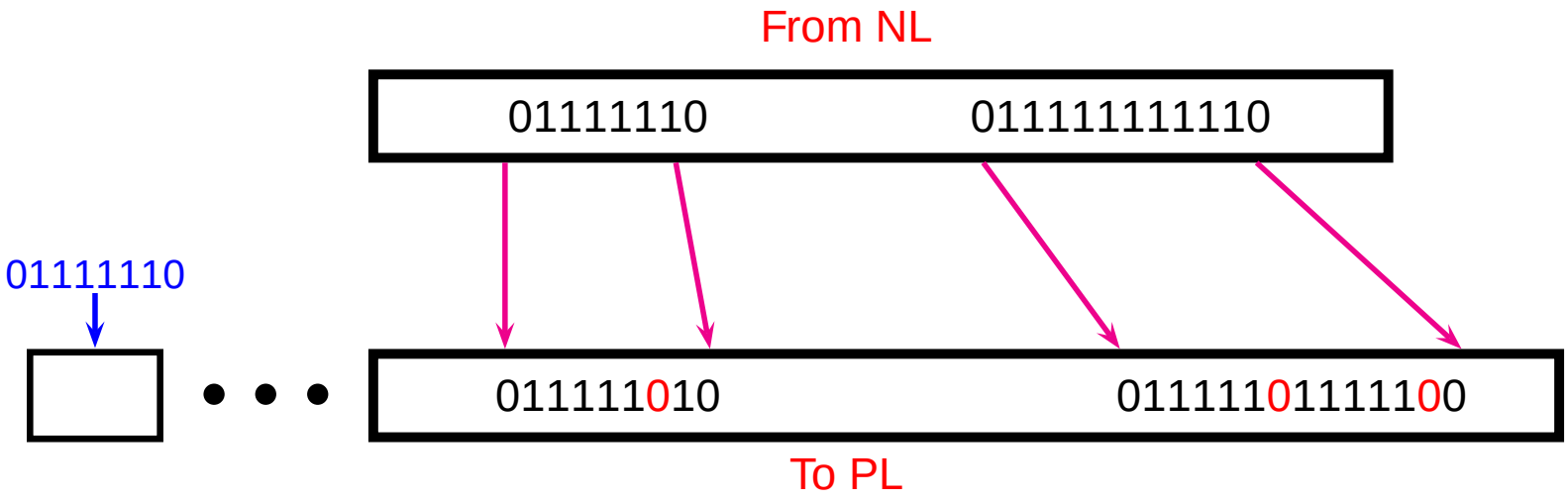
Bit stuffing used mainly in bit–oriented protocols calls for the insertion of extra bits into undesired data bit sequences.

Escape



Stuffing

If the frame delimiter is 01111110 it is common to require that the sending DLL **stuffs** (inserts) a 0 after every 5 consecutive data 1s, so that the only possible sequence of 6 1s is the flag sequence.



Several variations exist; they all are based on the same basic idea.

Link control

Three terms are used in describing the role of the DLL:

Error control: Error control is done in every layer up to layer 4. Layer 2 (DLL) is responsible for error correction which comes in three forms:

1. **BEC**, called **ARQ** (Automatic Repeat Request) when done by DLL.
2. **FEC** (when possible). This task is done jointly by the PL and the DLL (no clear boundary).
3. Ignoring the incorrect frames or replacing them with earlier frames (this works for synchronous media, such as video).

Flow control: to prevent the sender from **overwhelming** the receiver (buffer overflow, wrong bit rate).

Link control: is the combined task of error and flow control.

Automatic Repeat Request

The frame header must contain a frame sequence number that can be used in a repeat request. The minimum length of the sequence number field is well researched but real–life protocols ignore this body of research and use either a 16–bit or a 32–bit field.

The basic idea is to have the receiver send back (to the transmitter) an **ACK** frame containing the sequence number of the frame expected next.

To take care of lost frames (which cannot be acknowledged because the receiver is not aware of their existence), a **timeout** scheme is added.

The ARQ mechanism is best described in terms of an abstract **sliding window** made of the range of frames that are of current interest.

Sliding window

The sender and the receiver maintain separate sliding windows:

Sender's window: is made of frames that are **outstanding**, i.e. sent but not known to have been received (no **ACK** came back).

Sender's pending window: is made of frames that are **pending**, i.e. could be sent but were not because they do not exist.

Receiver's window: is made of slots to be filled by frames that have not been successfully acknowledged yet. The simplest case is a window of size 1: a buffer for the next frame; this will do for all protocols not using selective acknowledgments.

The size of the Sender's Window reflects the two aspects of link control:

- Outstanding frames are kept for error control. The window is made of frames *in transit* that can only be discarded after they are acknowledged.
- Pending frames are empty frame slots waiting for submissions from the NL.
- Not–in–window frames are existing frames that cannot be sent because of the limits imposed by flow control.

One frame can be in a transition state between being outside the window and inside the window when it is in the process of being sent.

Variants of sliding window protocols

Blast: The sizes of both windows are infinite.

Stop–and–Wait: The sizes of both windows are 1.

Go–Back–N: The size of the Send Window is N and the size of the Receive Window is at least 1 and no more than N.

Selective Acknowledgments: The size of the Send Window is N and the size of the Receive Window is M where $1 < M \leq N$.

Negative Acknowledgments: Same as the previous case, differs in the interpretation of the acknowledgment frames.

The BLAST protocol

The sender transmits without any consideration for the receiver. This protocol works if the receiver's buffer space is unlimited.

Obviously it is widely used in synchronous media transmissions because the sender must empty its buffers in a synchronous way.

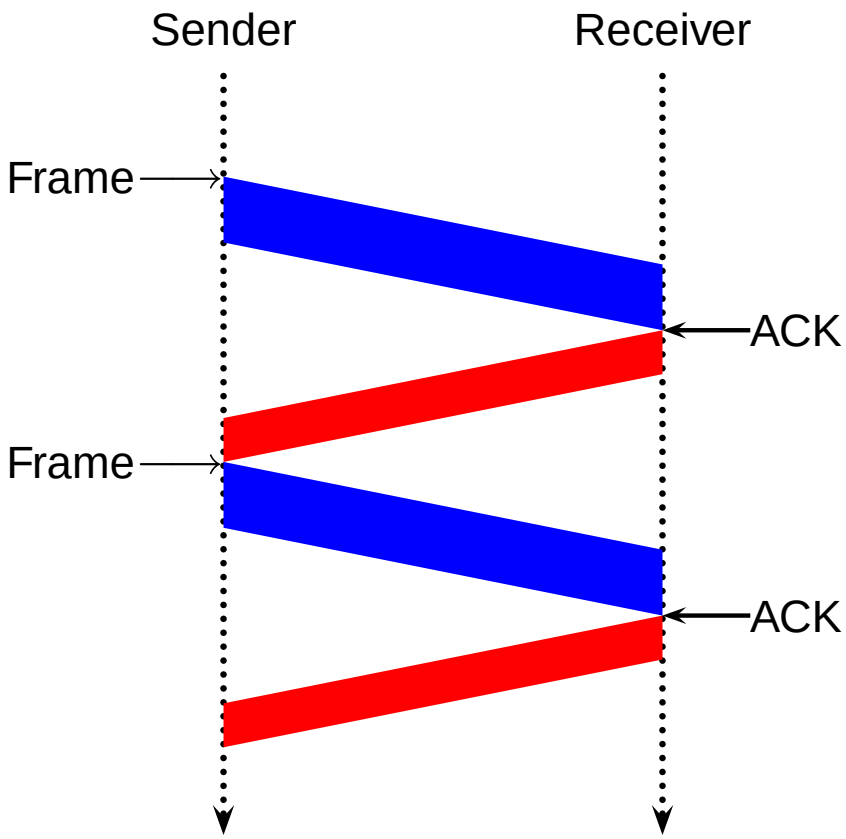
Otherwise it has no merit.

Stop–and–Wait

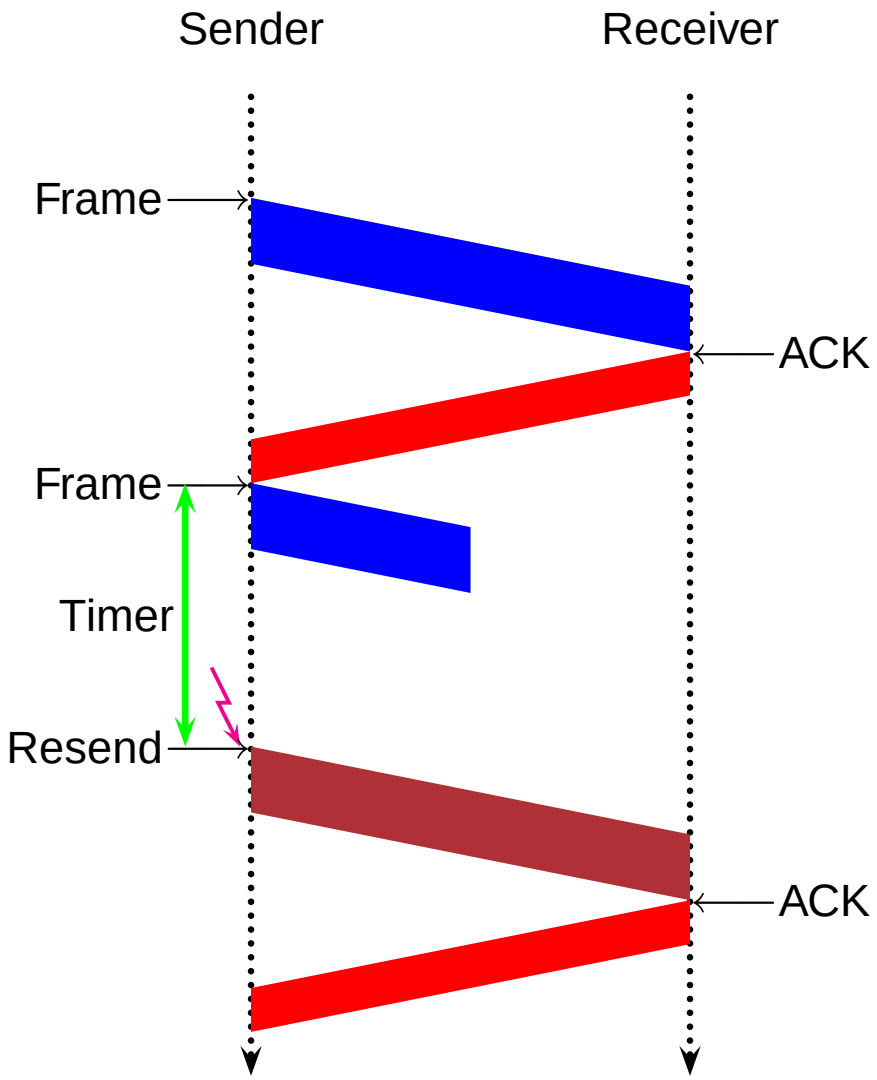
This is the simplest protocol with feedback. The sender's window being of size 1, the sender can send only 1 frame and must then wait for an acknowledgment.

IF an acknowledgment comes, the sender slides its window by one frame and continues on. If a timeout occurs before the acknowledgment arrives, the sender retransmits the outstanding frame and waits for an acknowledgment.

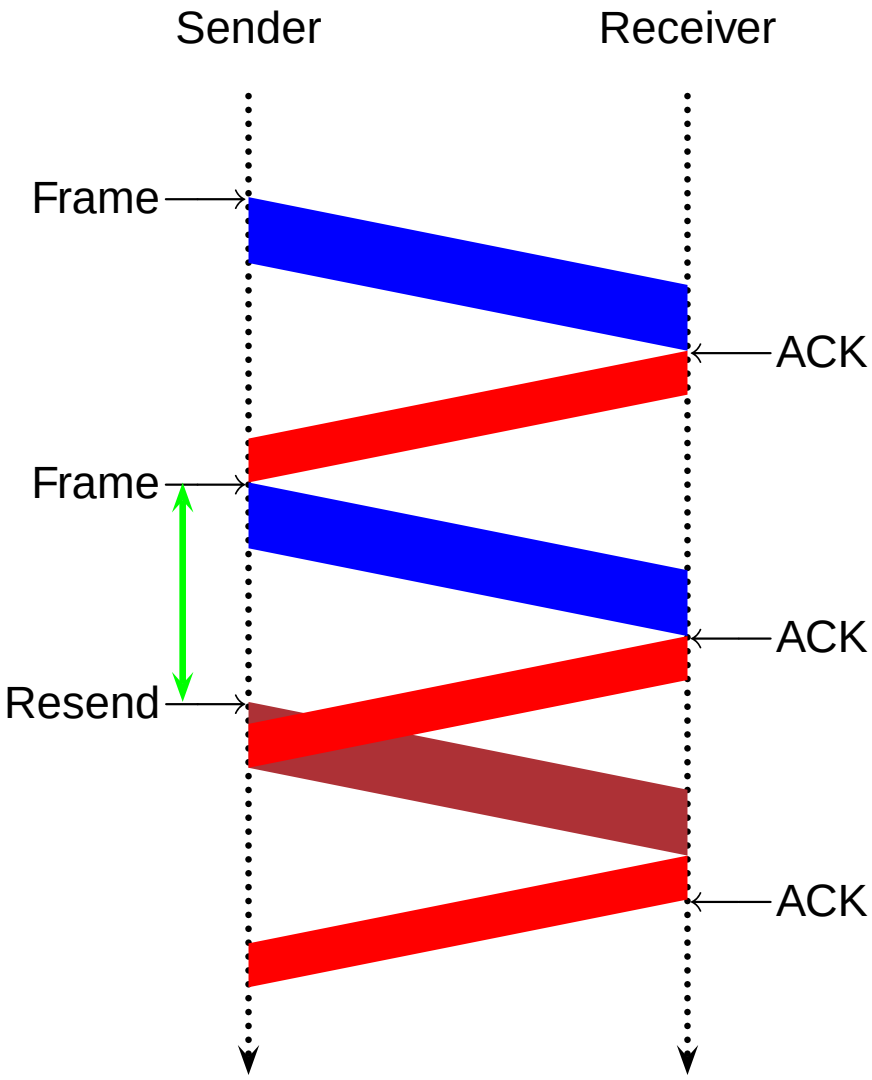
Stop–and–Wait in a noiseless channel



Stop–and–Wait in a noisy channel

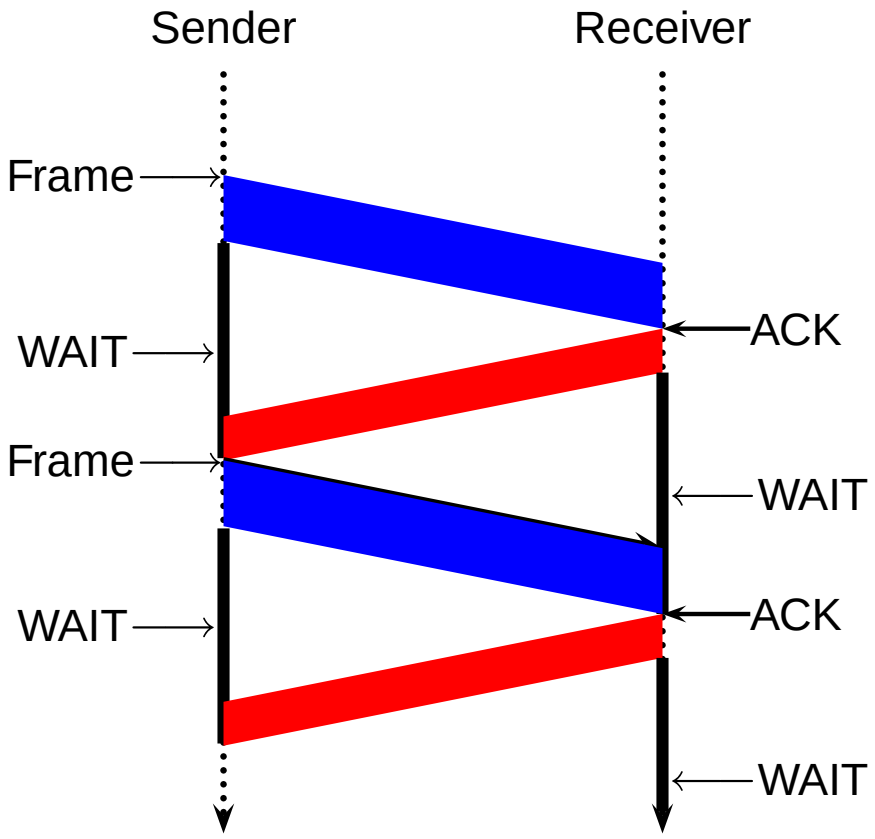


Another Stop–and–Wait scenario



The length of the timer fuse is an important parameter: too short causes unnecessary retransmissions; too long reduces the throughput of a noisy channel.

The only drawback of Stop–and–Wait is its inefficiency.

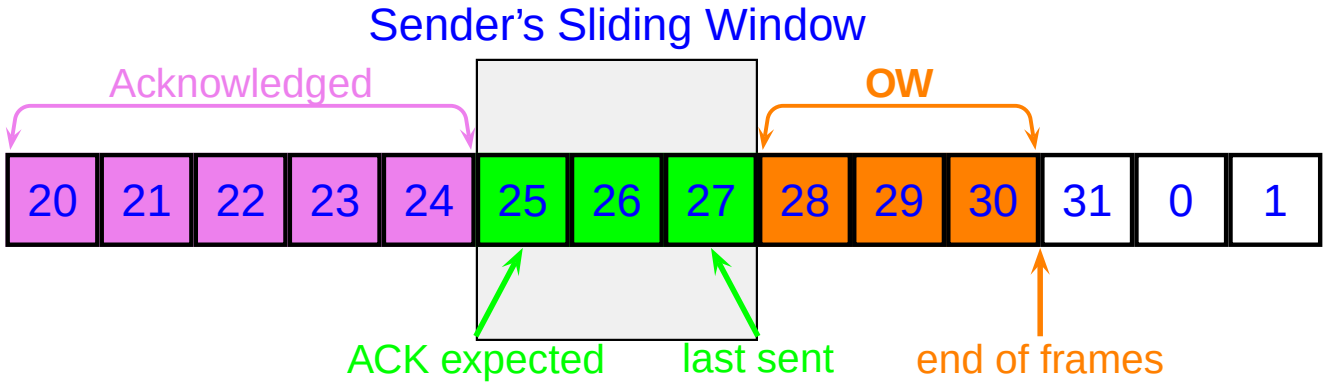


An obvious improvement is to allow pipelining, i.e. sending several frames while acknowledgments are moving in the reverse direction.

Go–Back–N

The Stop–and–Wait protocol is a special case of the Go–Back–N for N = 1.

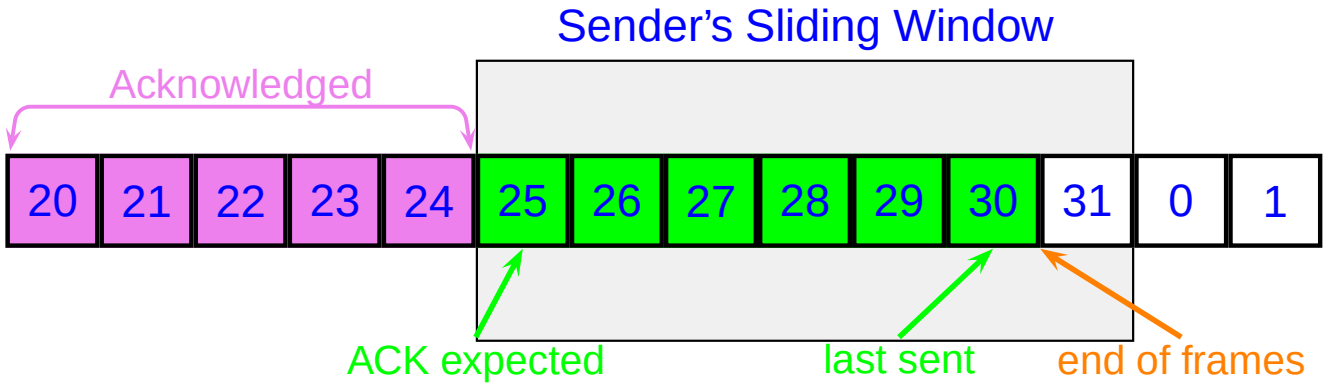
Consider a Go–Back–3 protocol. The sequence number field is 5 bits long, so the sequence numbers range from 0 to 31.



Frames 31, 0, and 1 do not exist yet; they will be created if more datagrams are submitted by the NL.

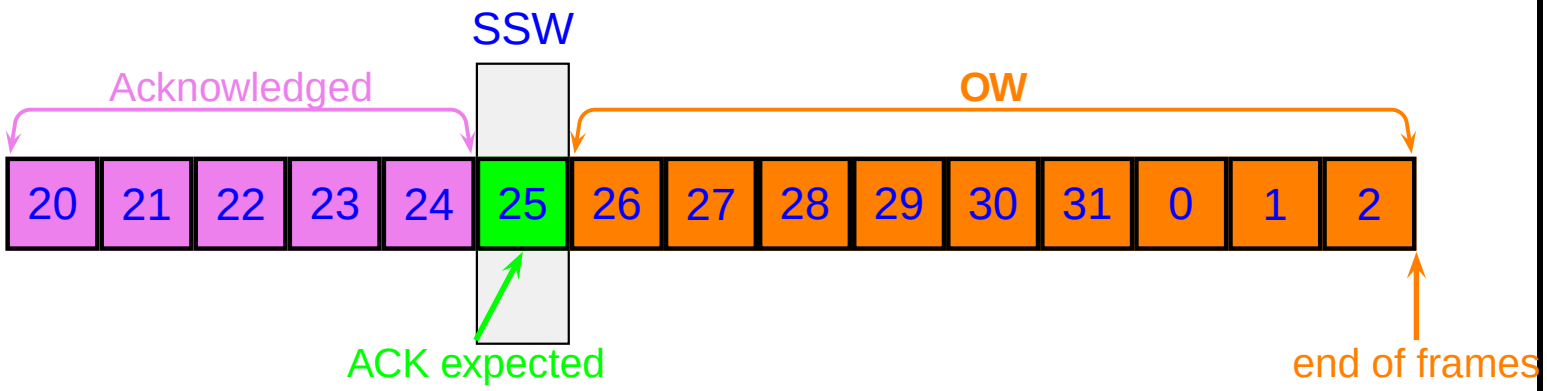
Go–Back–N

Another possibility; this time the receiver has lots of buffer space; knowing this (see **TCP**) the sender expanded the window to 7, but currently does not have enough frames to fill the window.

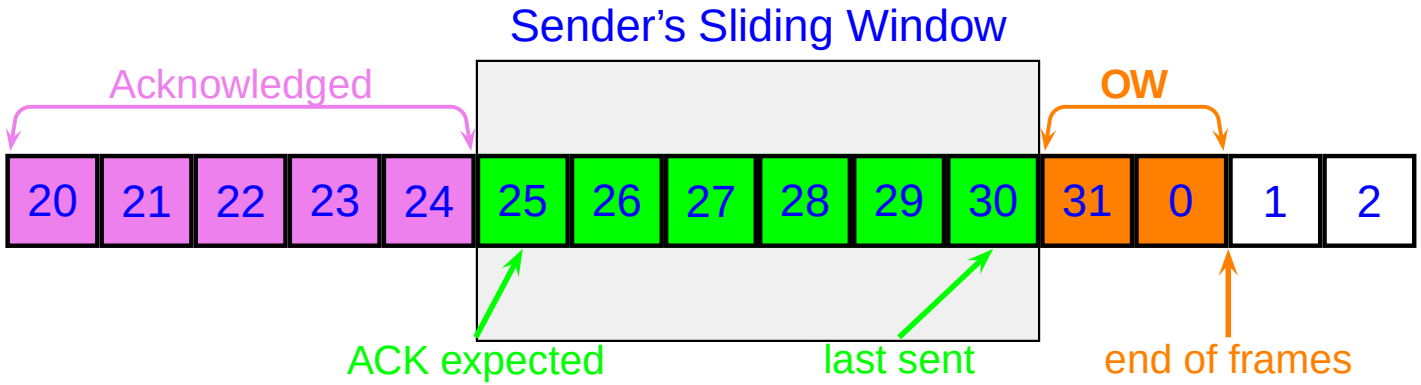


Go–Back–N

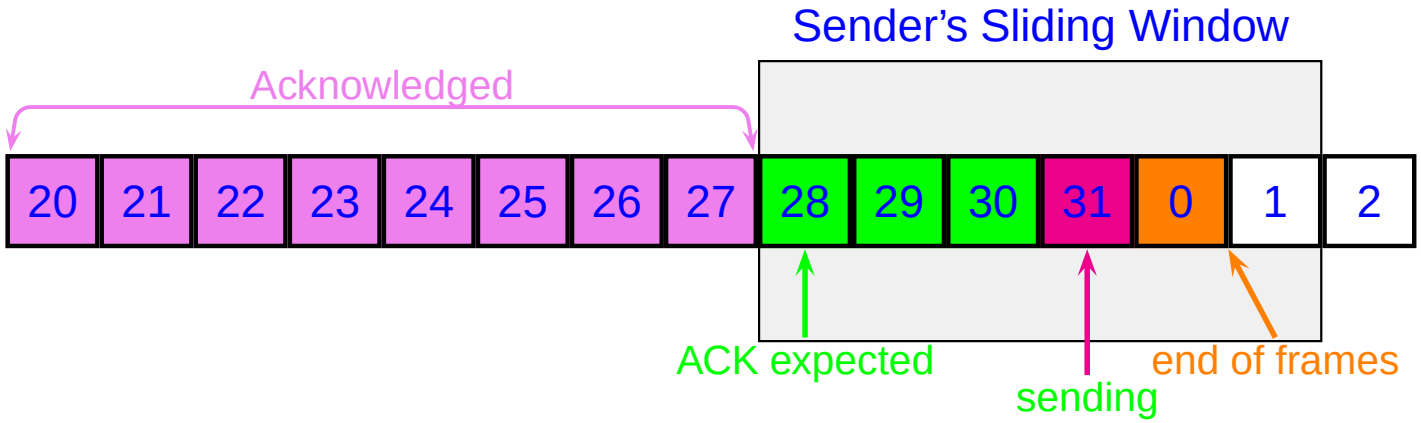
Another possibility; this time the sender has plenty to send but the receiver is facing a buffer overrun and asked to slow down.



Go–Back–N



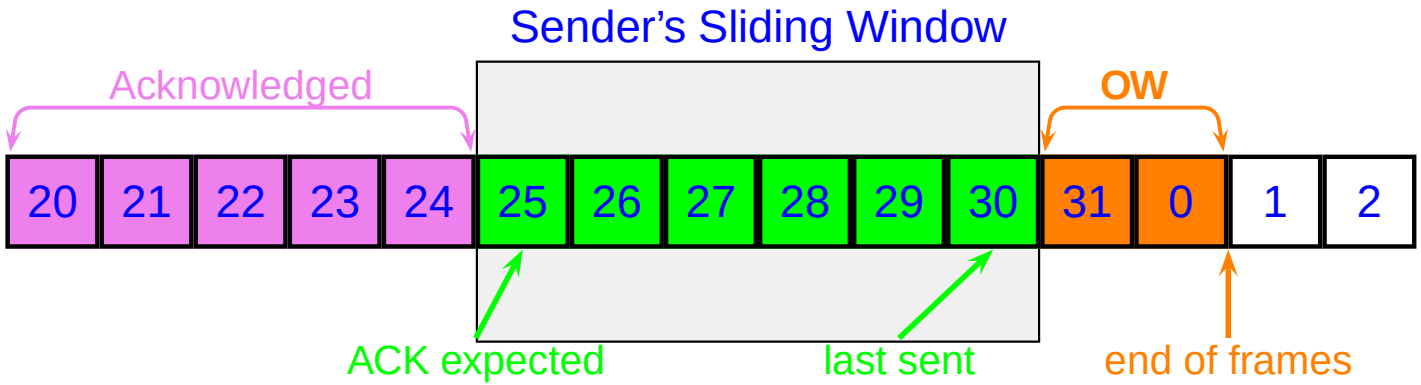
An **ACK** arrived with the sequence number 28 (implying that 28 is expected next).



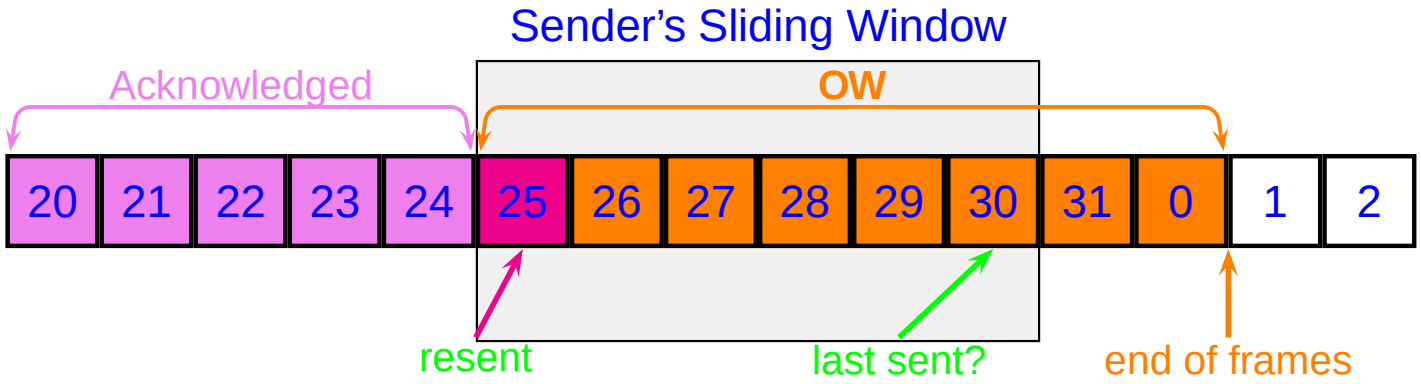
Timer

A timer fuse is set whenever a frame is sent in full. Obviously, the oldest ticking fuse expires first; it is associated with the frame waiting for an acknowledgment.

When that happens, the sender retransmits the whole window:

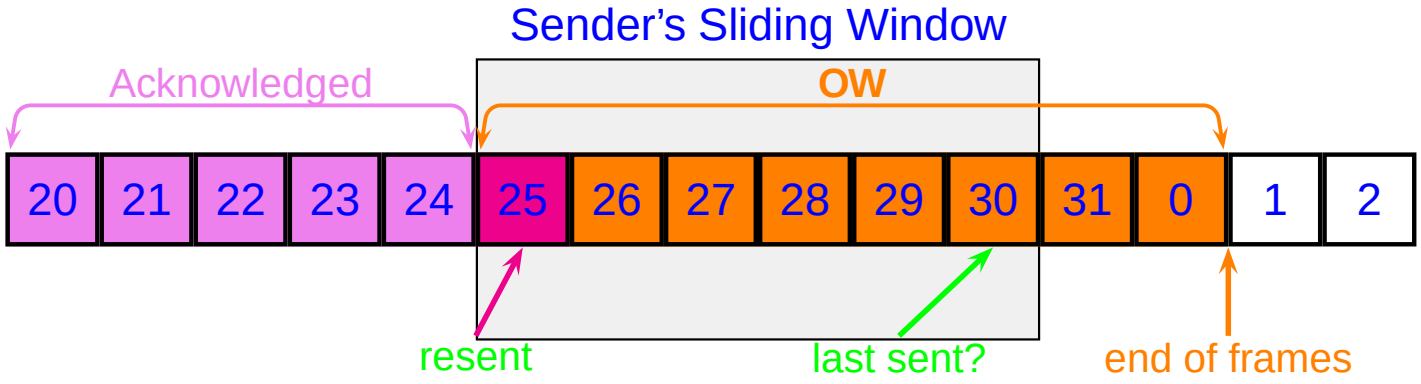


After the timeout:

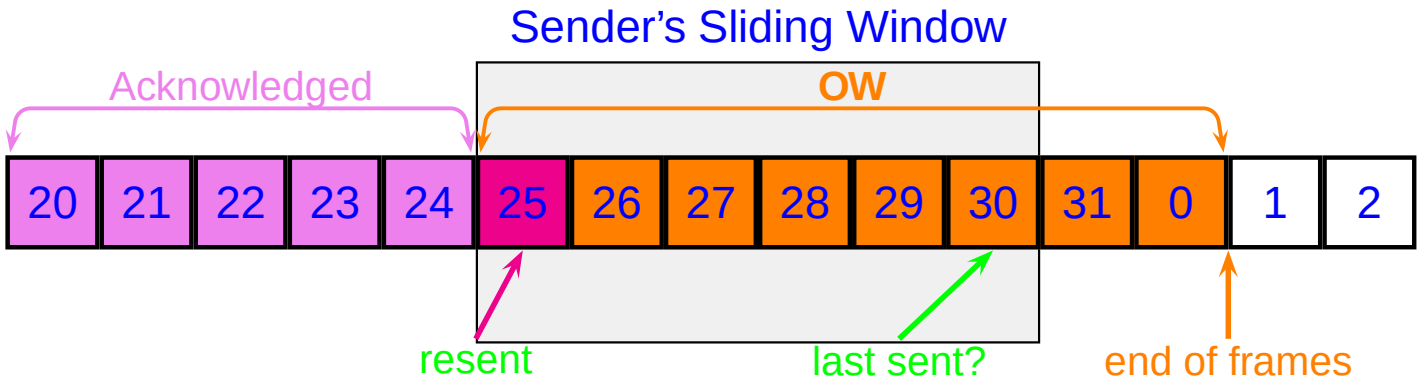


Situation during retransmission

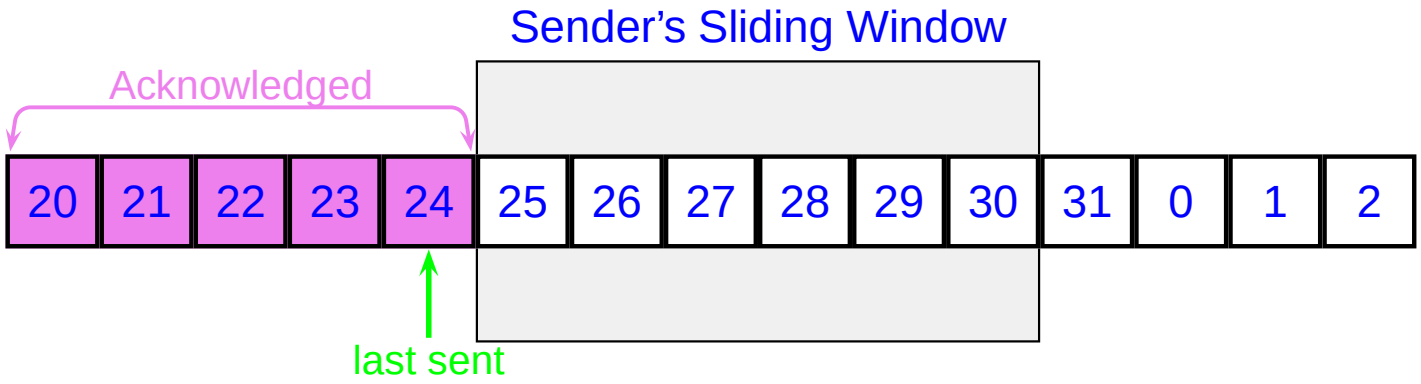
A timeout starts a procedure called **Fast Retransmit**. All the unacknowledged frames that are inside the window are retransmitted one after another. This causes a temporary inconsistent state.



What to do if an ACK 28 arrives at this very moment? Officially the frames 25–27 have not even been sent and yet they already are acknowledged. The outcome depends on the implementation.



Note that the situation above differs little from the situation when in the following state the DLL was suddenly handed 8 frames by the NL.



Receiver in Go–Back–N

The receiver's behaviour is simple: acknowledge every frame that is correct; ignore everything else.

```
S = isn ;
while( 1 ) {
    wait() ;  // will be interrupted by PL's interrupt
    while( Framecount > 0 ) {
        Frame = GetFrame() ;
        if( !Corrupted( Frame ) && Frame→seq == S ) {
            Accept( Frame ) ;
            S++ ;
            Acknowledge( S ) ;
        }
        Framecount−− ;
    }
}
```

S is the expected sequence number. `Framecount` is needed in case the sender manages to deliver more than one frame while the inside of the while loop executes (there

should be some form of mutual exclusion around `Framecount`).

Selective acknowledgments

When selective acknowledgments are used, the receiver's window must be of the same size as the sender's window.

There are two ways to use selective acknowledgments:

- Send an **ACK** frame with a list of frames that are acknowledged.
- Send both positive and negative acknowledgments. The timeout mechanism can become quite elaborate in this scheme.

Both approaches rely on a fundamental property of a direct link between a sender and a receiver:

> *Frames must arrive in the same order in which they were sent, if they arrive at all.*

Selective acknowledgments

The receiver uses an algorithm similar to the one used in Go–Back–N but acknowledges any correctly received frame, even if its sequence number is not consecutive. Unlike "normal" **ACK**s, the acknowledgment specifies the frame number of the frame that was correctly received (not the one expected next).

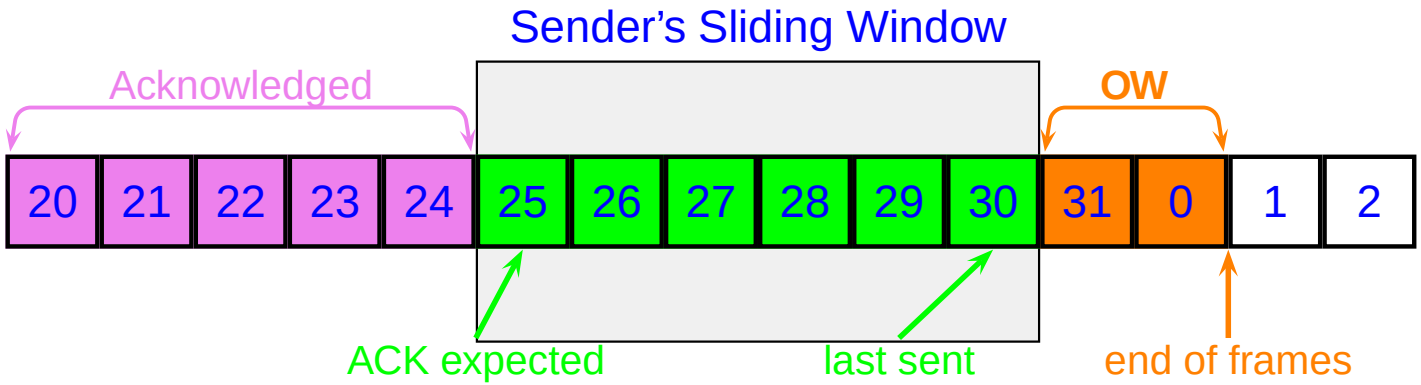
Note that acknowledging an out–of–sequence frame is an implicit negative acknowledgment for all the frames with preceding sequence numbers that were not acknowledged earlier.

Example: the sender receives the following sequence of acknowledgments:

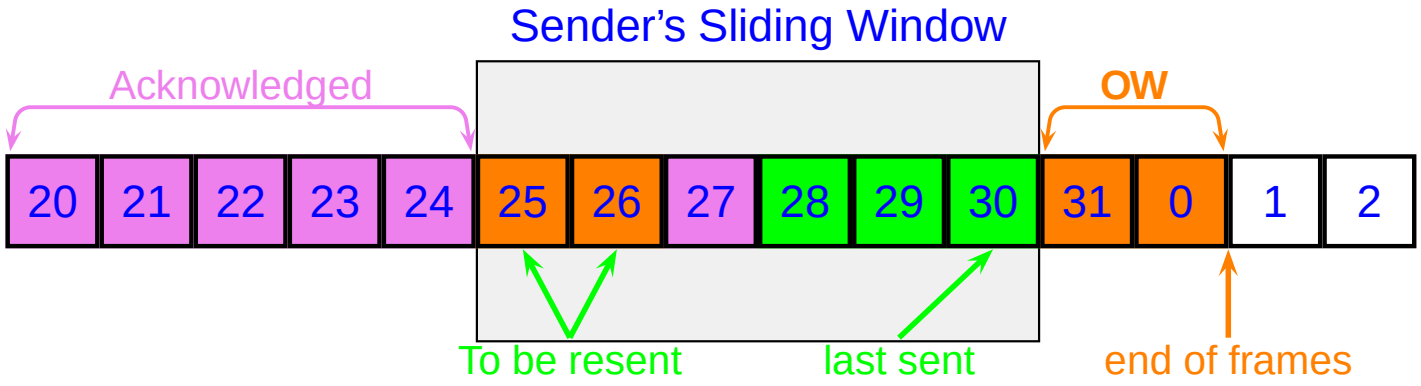
21 , 22 , 23 , 26

Clearly, frames 24 and 25 must have been either lost or rejected (if they arrived, they did so before 26 arrived).

Selective acknowledgments



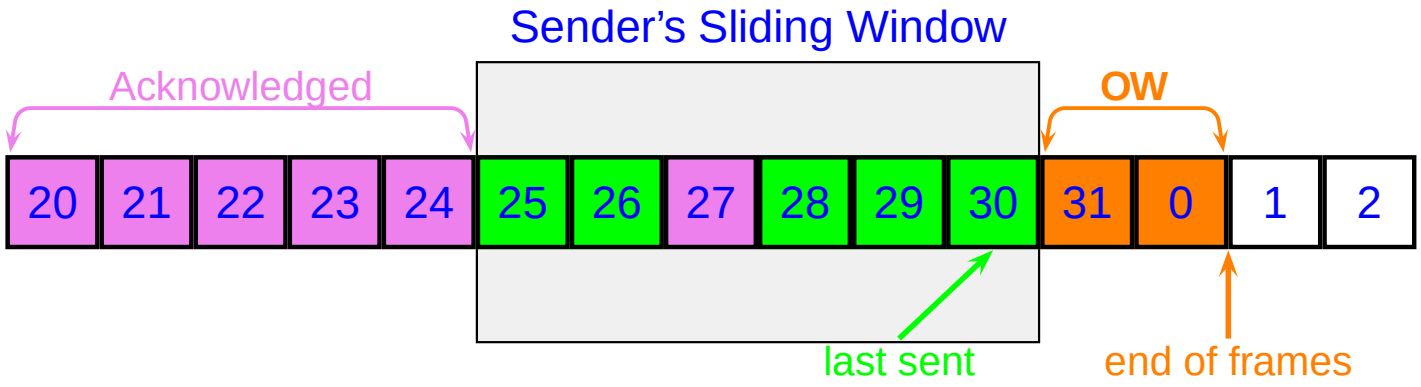
An **ACK** arrived with the sequence number 27 (stating that 27 was correctly received).



Timers

The sender must have a separate timer ticking for each outstanding frame. when a timer expires, only one frame is resent—the frame corresponding to the expired timer.

Suppose the timer corresponding to frame 29 expired in this situation:



There is no way to tell whether frame 25 was resent before frame 29 was sent or after. One cannot even tell how many times frame 25 was transmitted so far. Frame 25 has its own timer, so there is no need to resend it now.

What about frame 28? Since it is green, it must have already been retransmitted (at least once) because it was originally sent before 29 was sent.

Tricky cases

Consider the following sequence of events as observed by the sender:

1. Frames 25, 26, 27, 28 were sent.
2. An **ACK** 27 arrived.
3. Frame 25 was resent (and 26 is marked for retransmission).
4. The timer of frame 28 expired before the retransmission of 26 started.
5. Frames 26 and 28 were resent (in some order). Subsequently, frame 29 was sent.
6. An **ACK** 28 arrived.
7. (At least one frame is resent at this point.)
8. An **ACK** 29 arrived.

What to do now?

Negative acknowledgments

The other method of using selective acknowledgments is to combine them with explicit negative acknowledgments.

The idea is to delay positive acknowledgments until the moment when a frame is ready to be delivered to the NL.

When the receiver accepts a frame as non–corrupted, it checks whether it can forward it to the reassembly module (which will deliver this frame to the NL). The reassembly module requires that frames are given to it **in order**; this rule has to be obeyed even if it turns out that no reassembly is needed in a given case.

- If all the frames preceding the new frame have already arrived, the frame is considered **ready** to be delivered to the NL. While the exact delivery moment may lie in a very distant future (reassembly), the DLL$\leftrightarrow$DLL interaction is done (with respect to this frame).
- If some earlier frame is missing, the frame is **not ready** to be delivered to the NL.

Note that even if reassembly is unnecessary and the NL is connectionless, the above remains true because of the fundamental principle of the network stack:

no layer may assume anything about the properties of another layer, above or below.

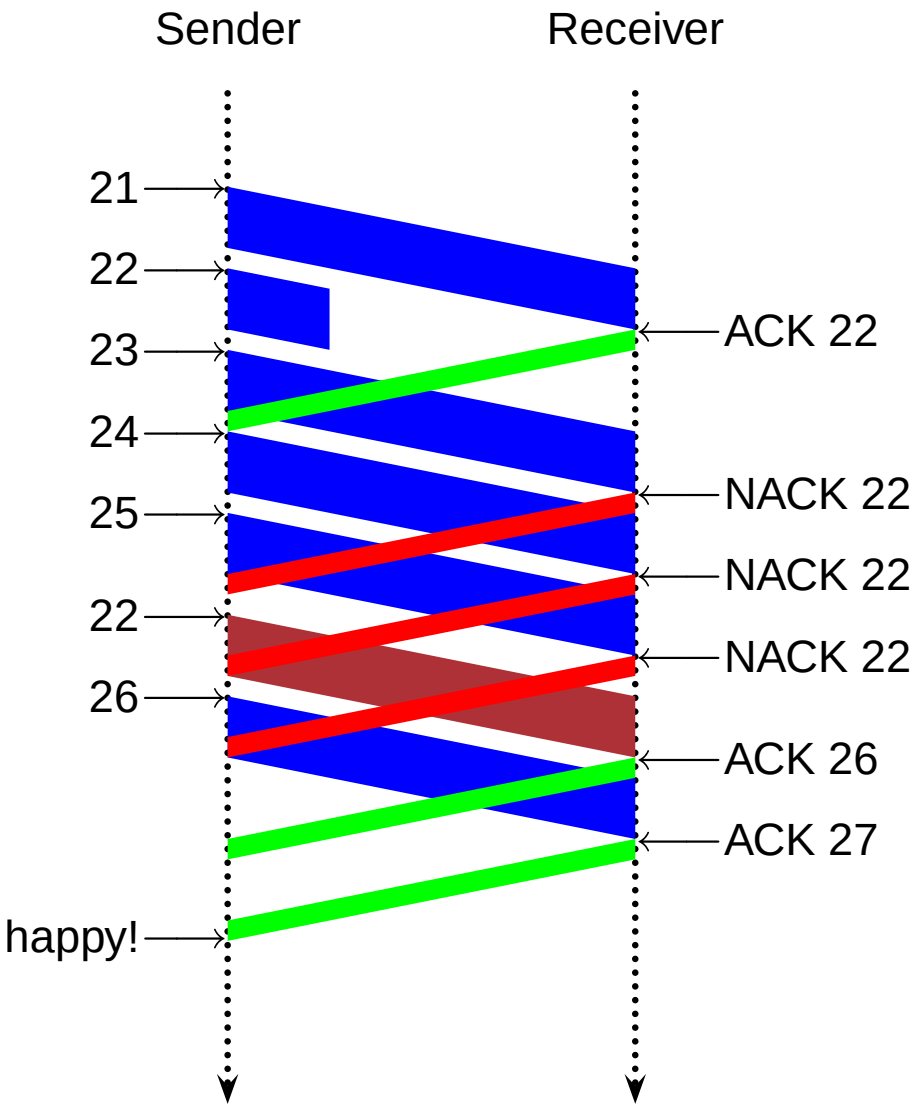
To ACK or to NACK

In principle, the receiver can react to every receiver frame by sending back a frame:

ACK: if the received frame is ready. The frame number in the ACK will indicate the frame expected next (the "normal" acknowledgment).

NACK: if the received frame is not ready. The frame number in the NACK will be the sequence number of the earliest missing frame.

In this example, the sender tries to send frames 21–26; frame 22 is destroyed in the channel. The receiver (±)acknowledges every frame.



Saving on the number of (N)ACKs

The numerous **NACK**s are not needed if the channel is not very noisy.

The first NACK should alert the sender and the rest will work properly provided that there are no further losses in this sequence. If there is another loss, expiring timers will force additional retransmissions (too many of them) guaranteeing a final success.

When NACKs are used, the timer fuse should be much longer, because its role is restricted to a last–resort backup for lost NACKs.

Flow control

The receiver may be slower than the sender. This can be caused by a number of factors, some of which are transient:

- The processor on the receiver's card is too slow in moving frames out of the input buffers.
- The receiver is fast enough but it is not able to deliver datagrams to the NL fast enough (the NL does not accept them fast enough, maybe because it is congested with other traffic). Undelivered frames clutter the DLL's buffer space.
- A malfunction or a DoS attack cause the receiver to be bothered with interrupts that must be handled, thus preventing the receiver from emptying its input buffers fast enough.

Whatever the reason, the receiver may face a situation called **buffer overrun** in which there is no space in the input buffer for newly arriving frames.

Flow control attempts to prevent buffer overrun by limiting the the maximum number of genuinely outstanding frames.

How to slow the sender

If the receiver is not fast enough, the sender must be told to slow down. This can be done in two ways:

- By withholding acknowledgments: the sender will find itself waiting for timeouts (sending nothing) and then by sending duplicate frames (which the receiver can partially ignore).
- By telling the sender to reduce the size of its sliding window.

The second approach, used in **TCP** at the TL, leads to dynamic window sizes.

Piggybacking acknowledgments

If communication is bi–directional, it is natural to incorporate acknowledgments in data frames sent in the opposite way.

the resulting protocols have to consider the option of delaying an ACK in order to piggyback it; another timer must be used to make sure that an acknowledgment is not delayed for too long (thus causing a timeout at the other end).