Transmitting without acknowledgments

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

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

The Automatic Retransmit Request mechanism was designed to allow the transmitter to send more than one segment before getting them acknowledged.

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

Sliding window

The sender and the receiver maintain separate sliding windows:

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

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

Receiver's window: is made of space to be filled by segments that have not been successfully acknowledged yet. The simplest case is a window of size 1 (MSS): a buffer for the next segment; 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 segments are kept for error control. The window is made of segments *in transit* that can only be discarded after they are acknowledged.
- Pending segments are empty segment slots waiting for submissions from the AL.
- Not–in–window segments are existing segments that cannot be sent because of the limits imposed by flow control.

One segment 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 $1 < \mathrm{M} \leq \mathrm{N}$.

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

The BLAST protocol

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

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

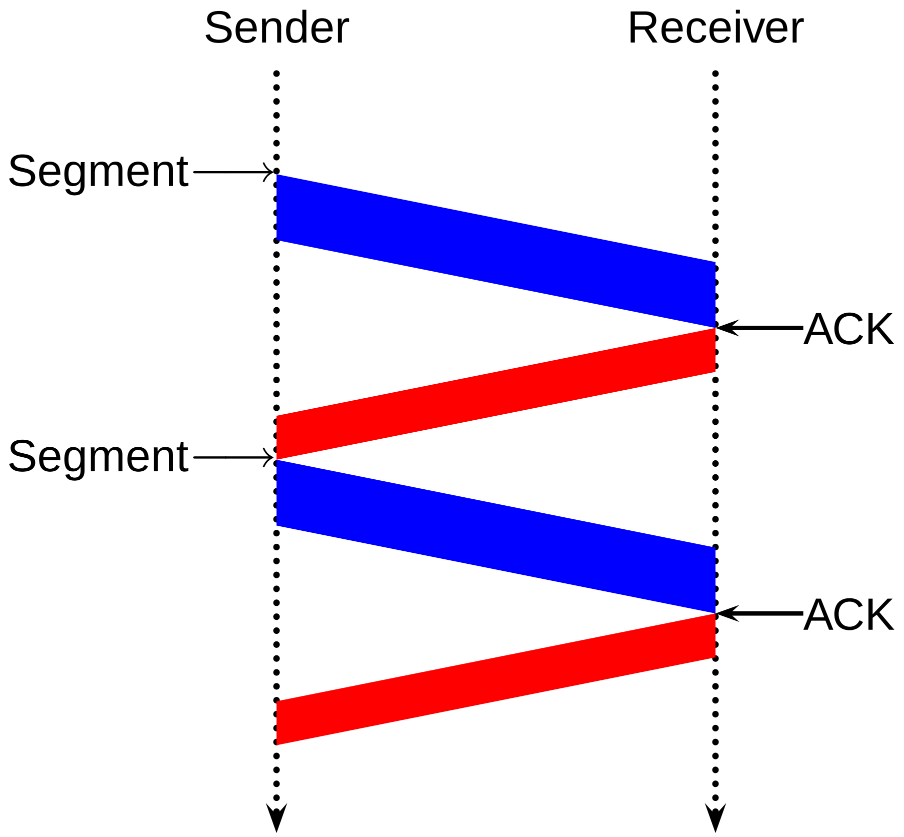
Otherwise it has no merit.

Stop–and–Wait

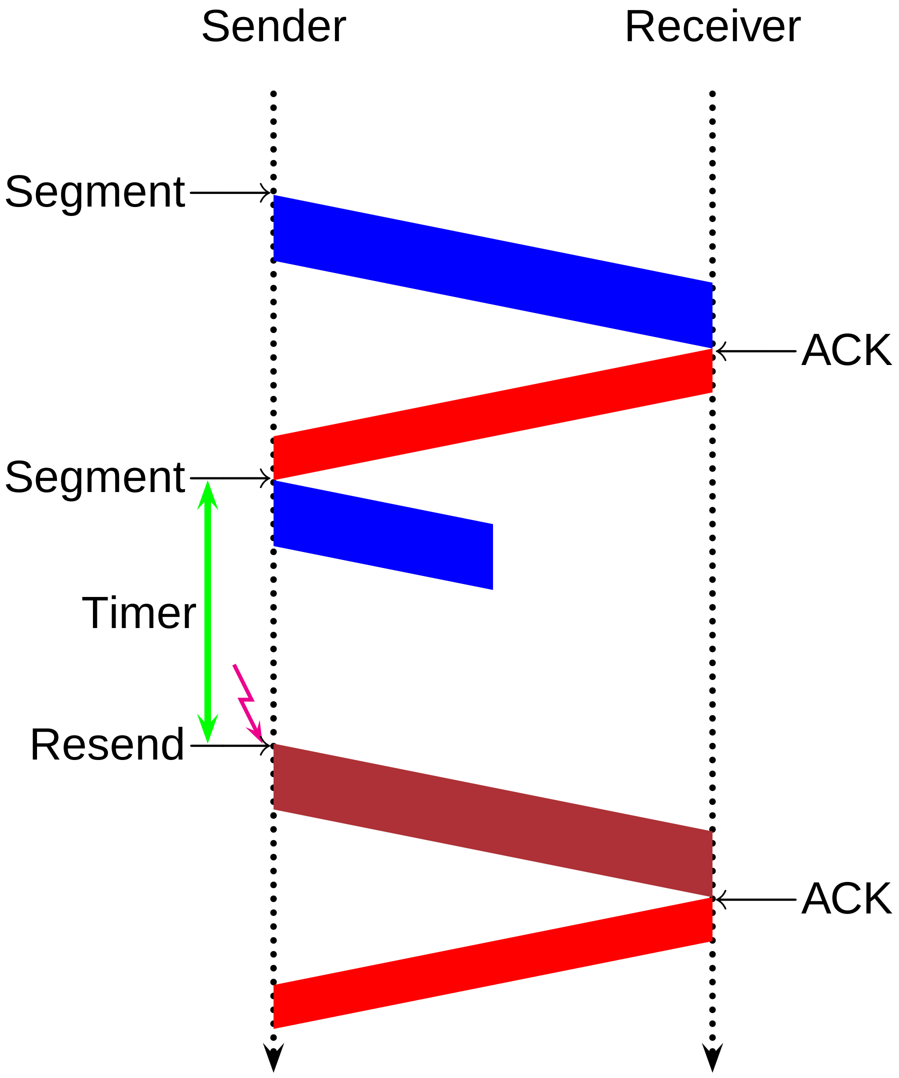
This is the simplest protocol with feedback. The sender's window being of size 1, the sender can send only 1 segment and must then wait for an acknowledgment.

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

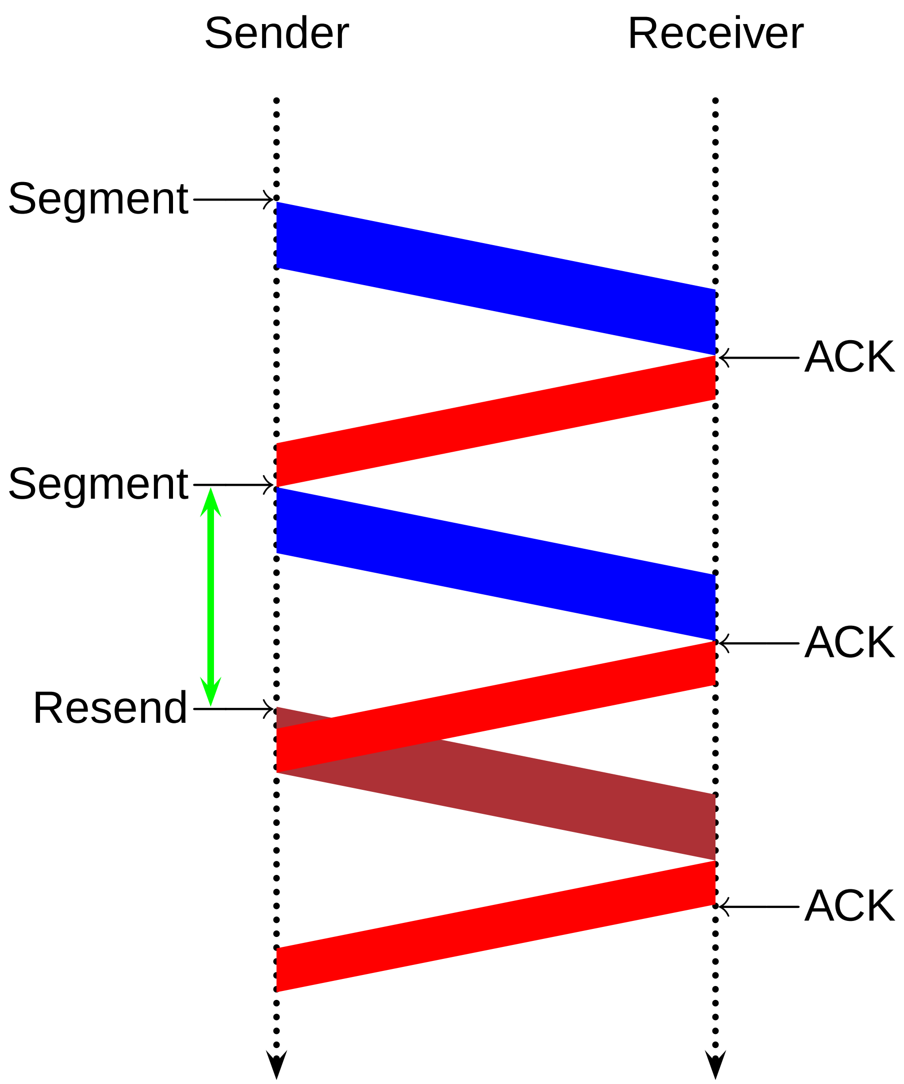
Stop–and–Wait in a noiseless channel



Stop–and–Wait in a noisy channel

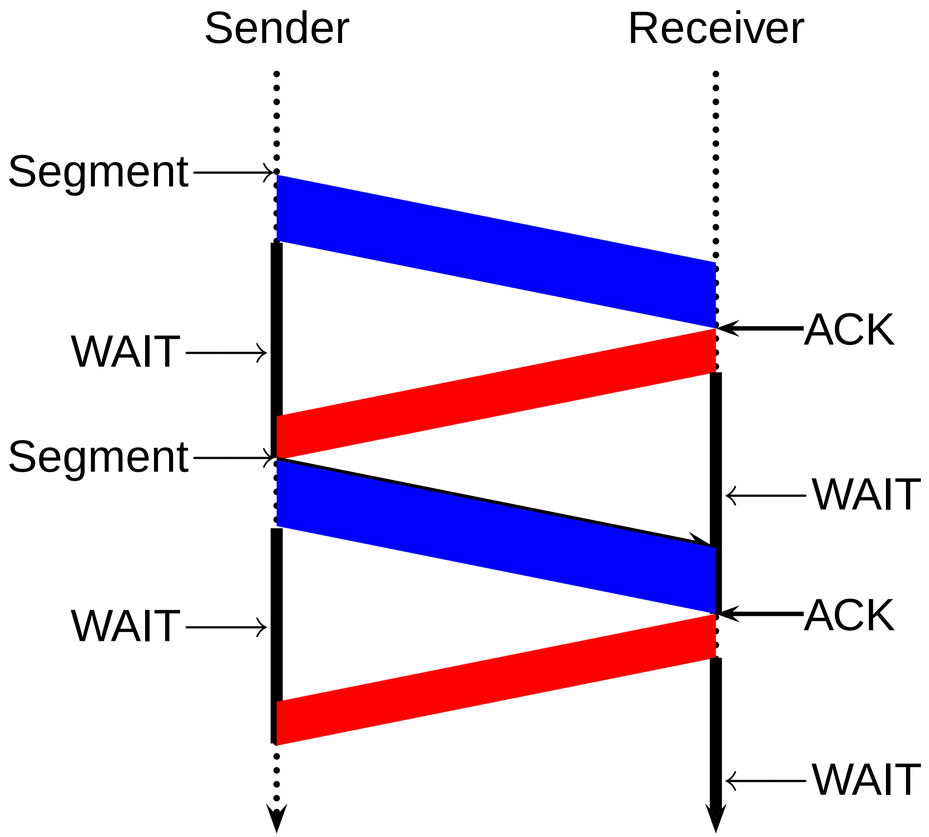


Another Stop–and–Wait scenario



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.

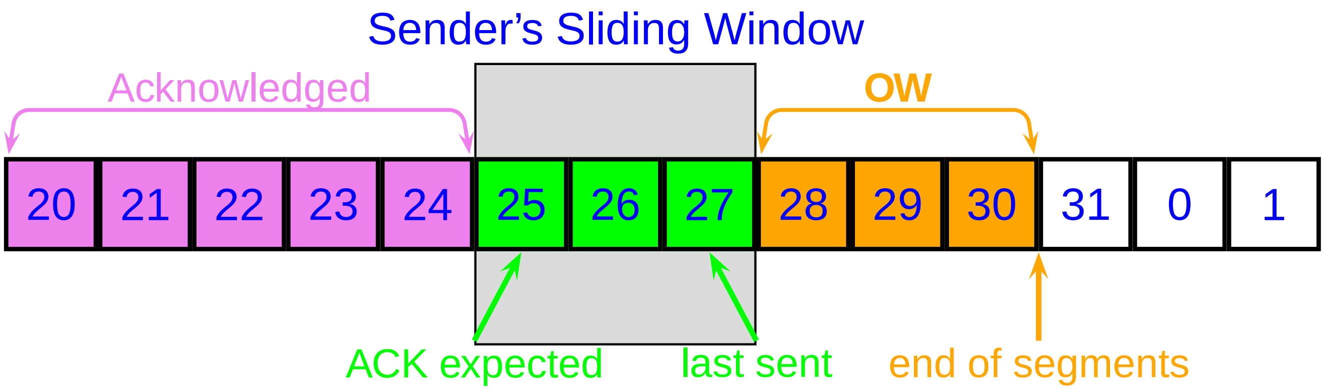


An obvious improvement is to allow pipelining, i.e. sending several segments while acknowledgments are moving in the reverse direction.

Go–Back–N

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

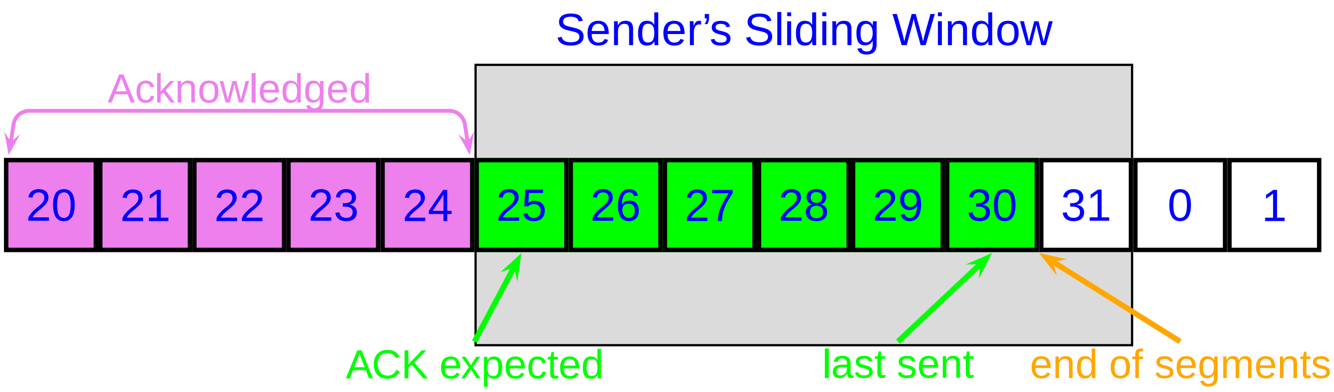
Consider a Go–Back–3 protocol. The sequence number field is 5 bits long (for illustration only), so the sequence numbers range from 0 to 31.



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

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 segments to fill the window.



Sender's Sliding Window

Acknowledged

20 21 22 23 24 25 26 27 28 29 30 31 0 1

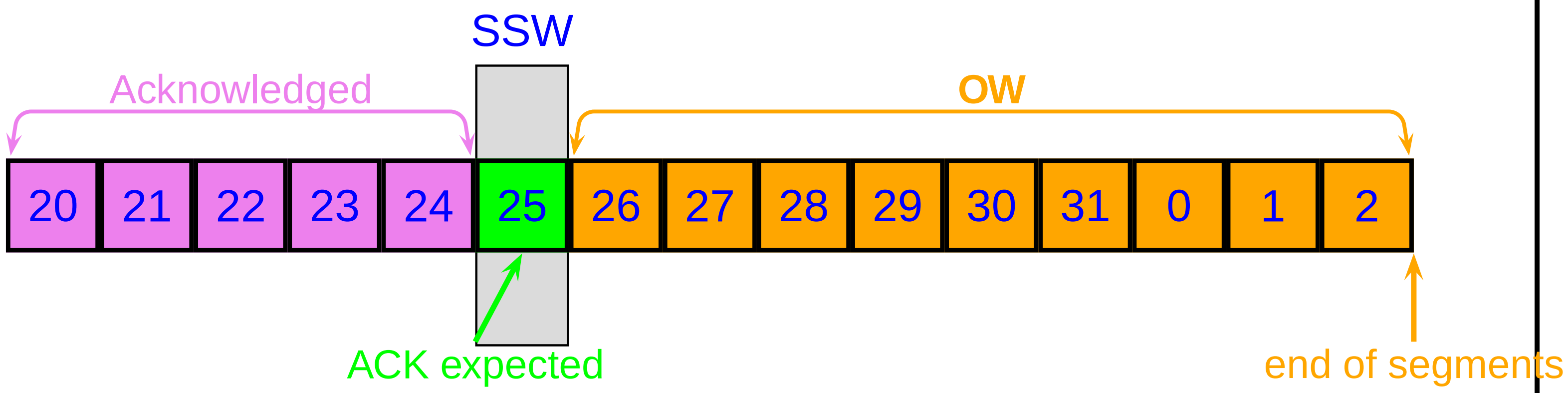
ACK expected

last sent

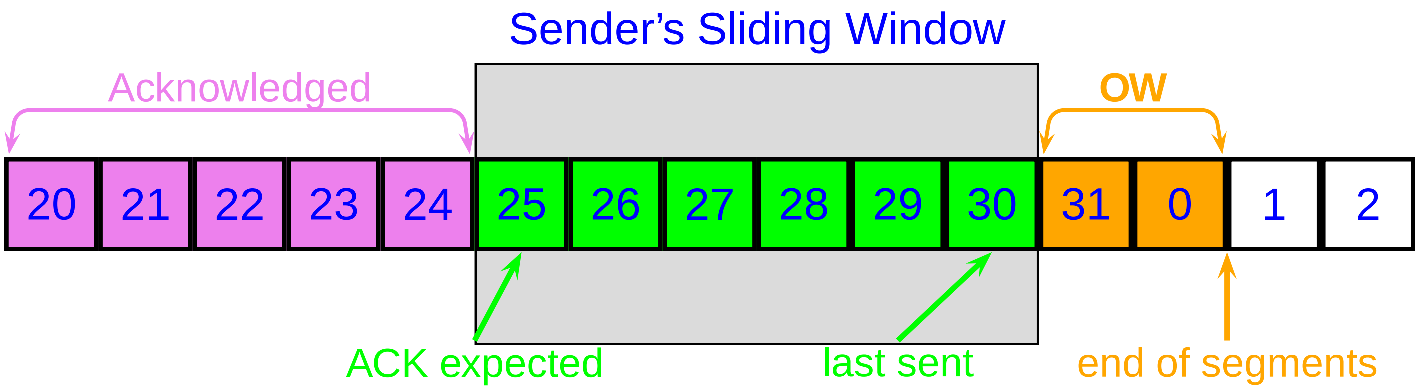
end of segments

Go–Back–N

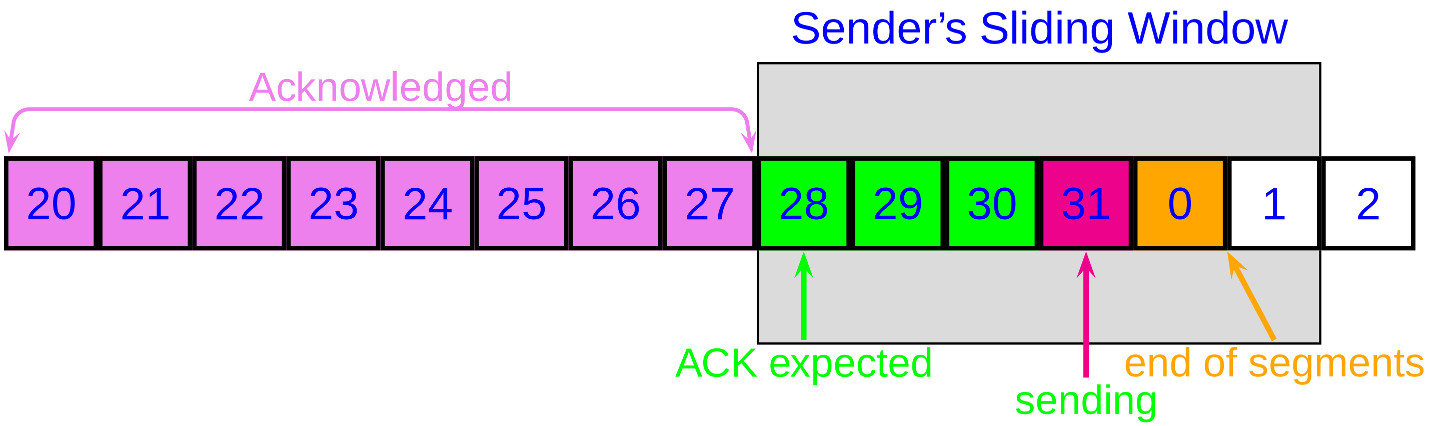
Another possibility; this time the sender has plenty to send but the receiver is facing a buffer overrun and asked to slow down.



Go–Back–N



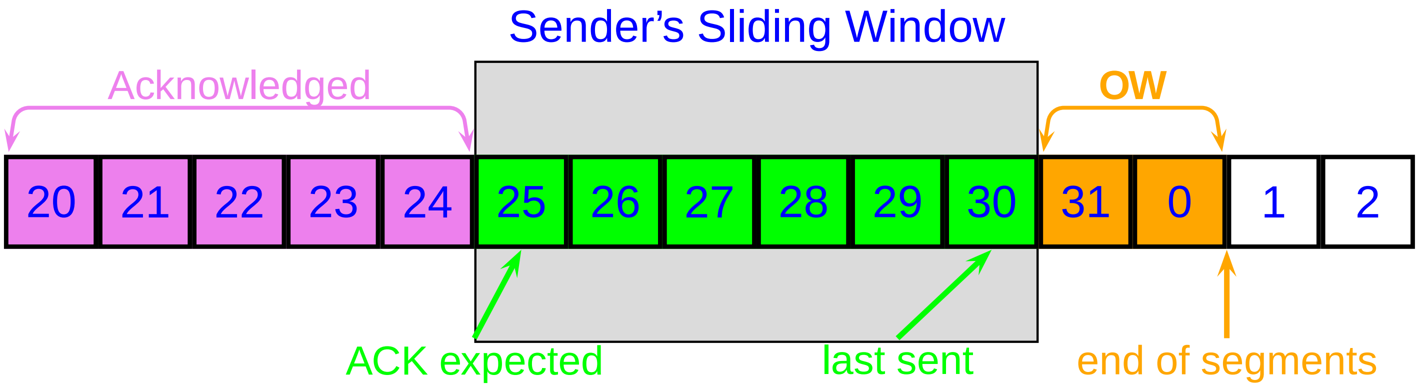
An **ACK** arrived with the sequence number 28 (implying that 28 is expected next).



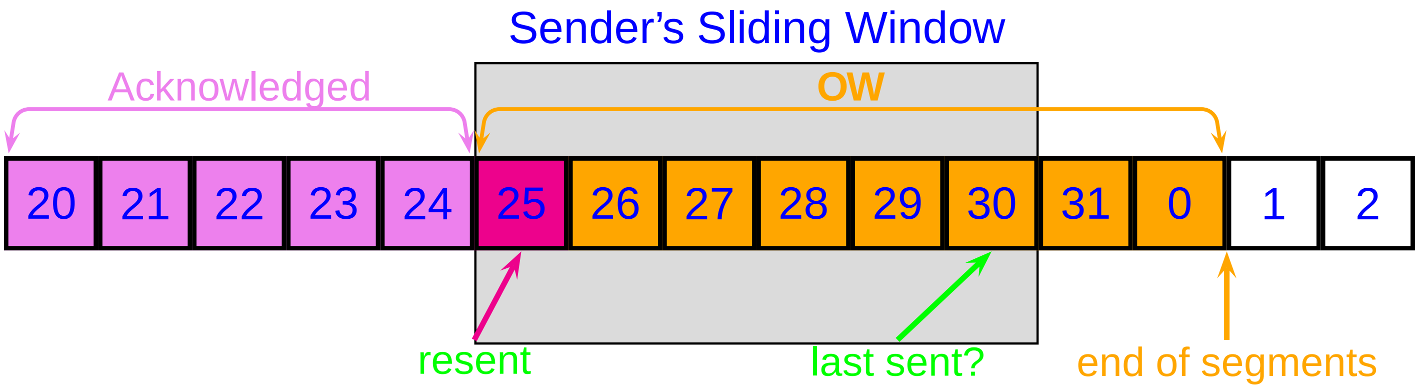
Timer

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

When that happens, the sender retransmits the whole window:

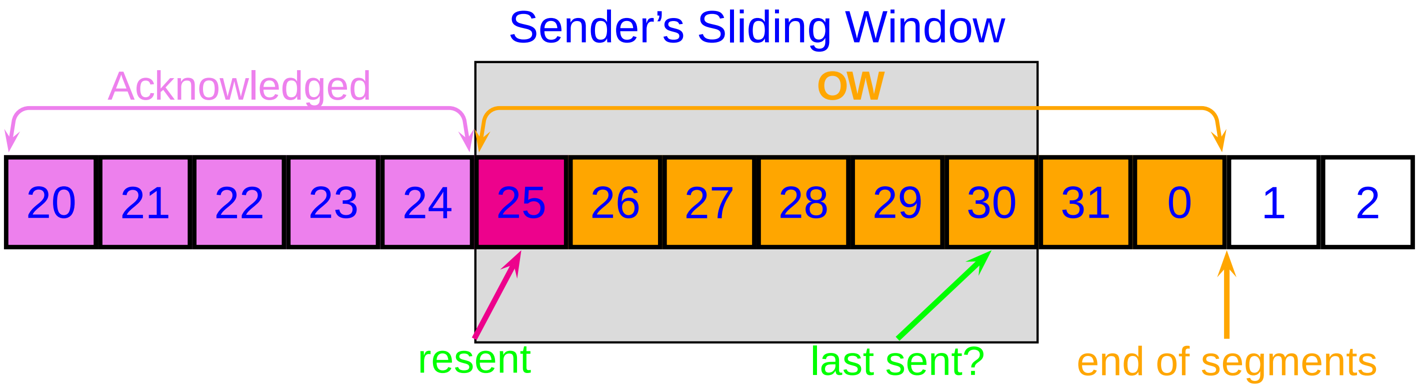


After the timeout:

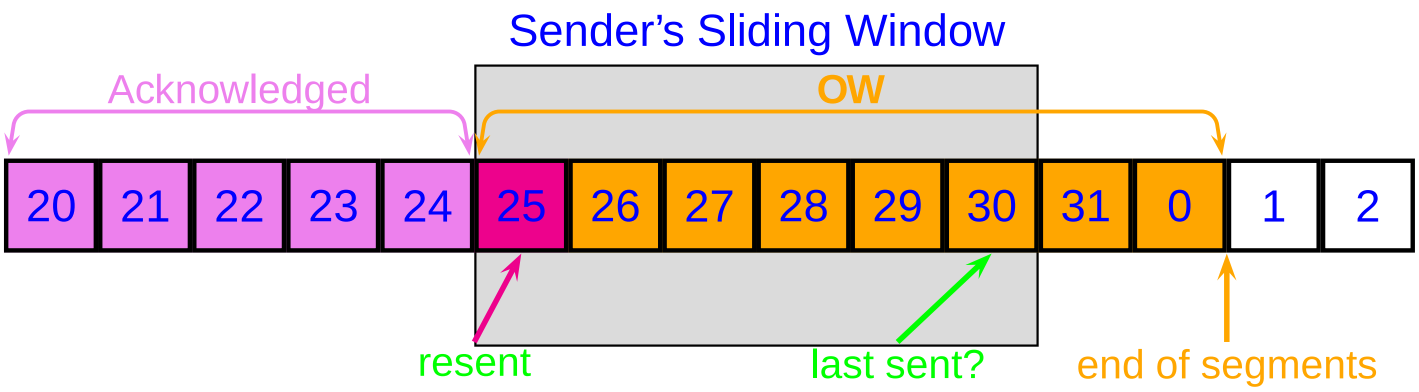

Sender's Sliding Window
Acknowledged
OW
20
21
22
23
24
25
26
27
28
29
30
31
0
1
2
resent
last sent?
end of segments

Situation during retransmission

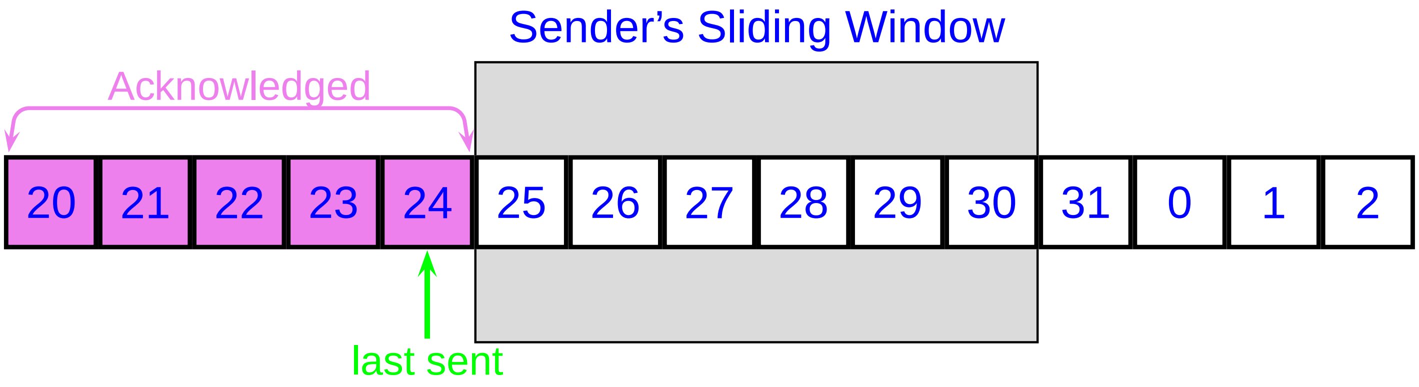
A timeout starts a procedure called **Fast Retransmit**. All the unacknowledged segments that are inside the window are retransmitted one after another. This causes a temporary inconsistent state.



What to do if an ACK 28 arrives at this very moment? Officially the segments 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 TL was suddenly handed 8 segments by the AL.



Receiver in Go–Back–N

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

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

S is the expected sequence number.

Segmentcount is needed in case the sender manages to deliver more than one segment while the inside of the while loop executes.

There must be some form of mutual exclusion around operations on **Segmentcount**.

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** segment with a list of segments 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 (TL handles a single direct virtual link):

> *Segments 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 segment, even if its sequence number is not consecutive. Unlike “normal” **ACK**s, the acknowledgment specifies the segment number of the segment that was correctly received (not the one expected next).

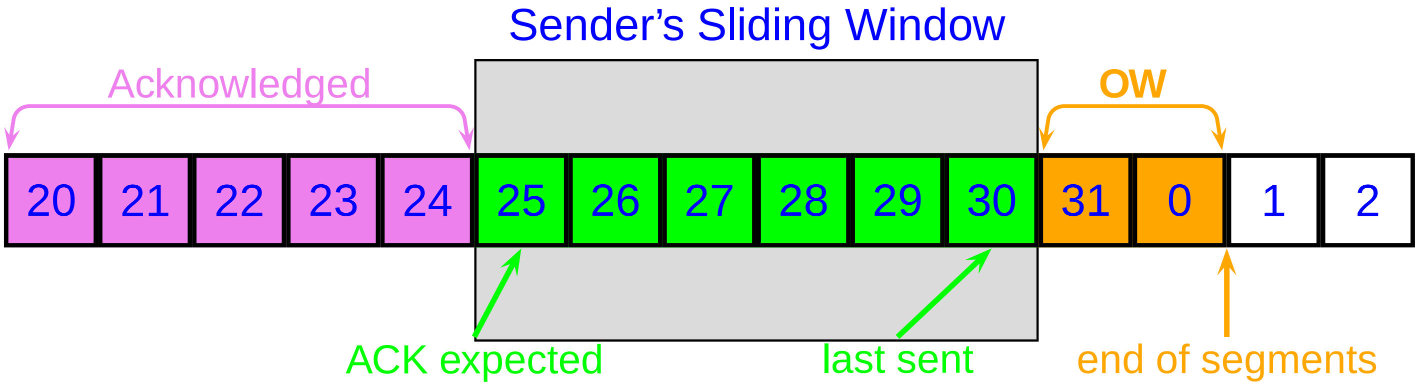
Note that acknowledging an out–of–sequence segment is an implicit negative acknowledgment for all the segments with preceding sequence numbers that were not acknowledged earlier.

Example: the sender receives the following sequence of acknowledgments:

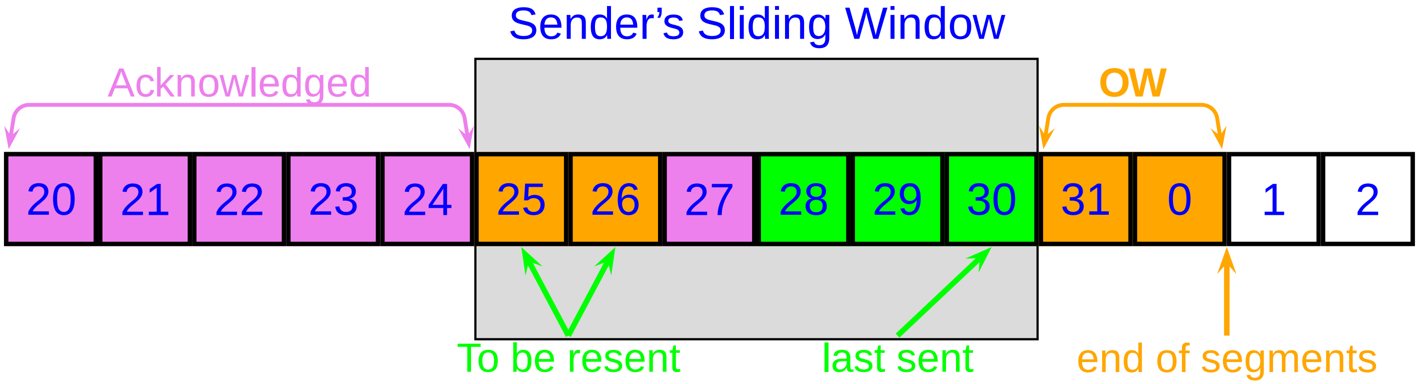
21 , 22 , 23 , 26

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

Selective acknowledgments



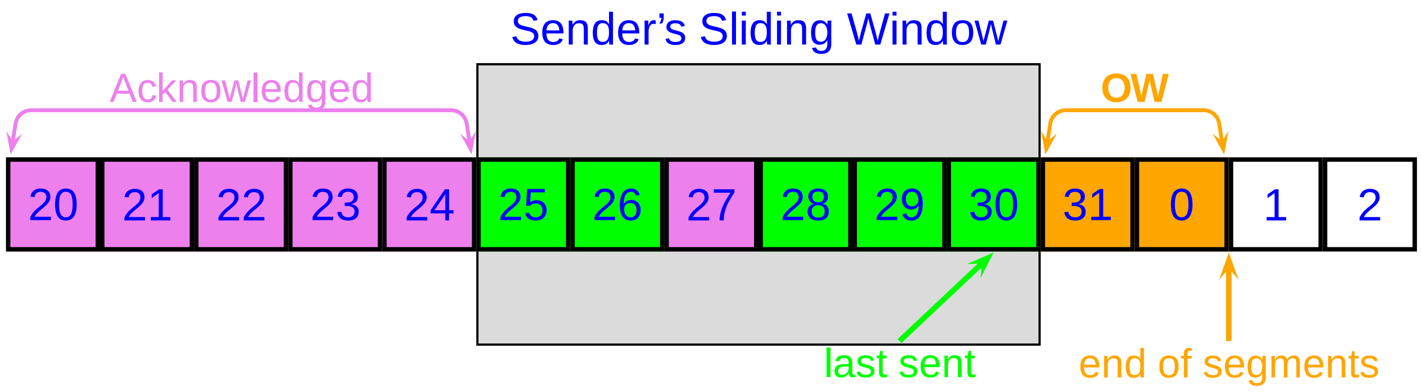
An **ACK** arrived with the sequence number 27 (stating that 27 was correctly received).



Timers

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

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



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

What about segment 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. Segments 25, 26, 27, 28 were sent.
2. An **ACK** 27 arrived.
3. Segment 25 was resent (and 26 is marked for retransmission).
4. The timer of segment 28 expired before the retransmission of 26 started.
5. Segments 26 and 28 were resent (in some order). Subsequently, segment 29 was sent.
6. An **ACK** 28 arrived.
7. (At least one segment 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 segment is ready to be delivered to the AL.

When the receiver accepts a segment as non–corrupted, it checks whether it can forward it to the AL.

- If all the segments preceding the new segment have already arrived, the segment is considered **ready** to be delivered to the AL. While the exact delivery moment may lie in a very distant future, the TL↔TL interaction is complete (with respect to this segment).
- If some earlier segment is missing, the segment is **not ready** to be delivered to the AL and a **NACK** should be sent, asking for the missing segment.

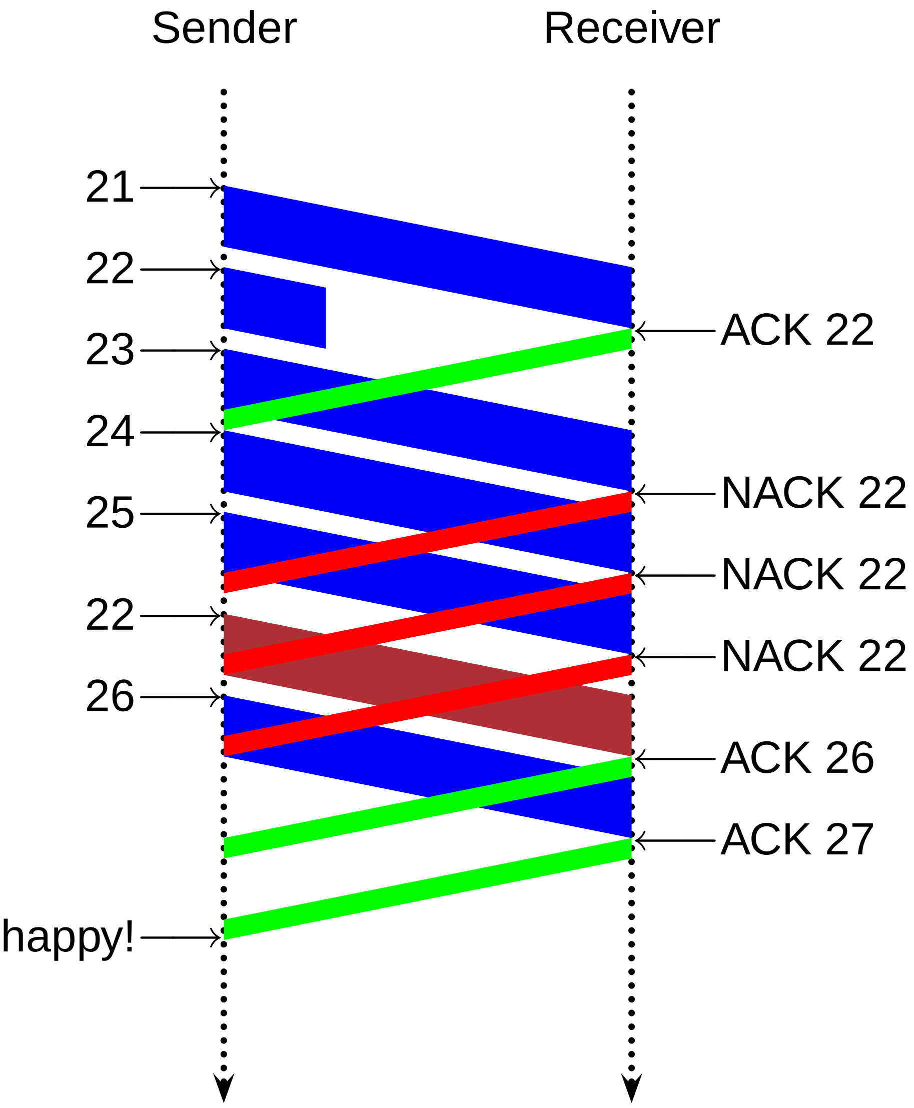
To ACK or to NACK

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

ACK: if the received segment is ready. The segment number in the ACK will indicate the segment expected next (the “normal” acknowledgment).

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

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



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 receiver's application does not ask for input.
- The receiver's is not fast enough in processing segments: the NL hands them at a faster rate than they are handled.
- A malfunction or a DoS attack cause the receiver to be bothered with other traffic 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 segments.

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

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 segments (which the receiver can partially ignore).
- By telling the sender to reduce the size of its sliding window.

The second approach is used in **TCP**; it leads to dynamic window sizes.

Piggybacking acknowledgments

If communication is bi–directional, it is natural to incorporate acknowledgments in data segments 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).