Where we are

- *protocol = format + rules*
- We need rules to define how the frames are handled:
  - How does sender know the frame reached the other end?
  - What does the receiver do when an error is detected?
  - What about “lost” frames?
  - How does the receiver prevent the sender from over-running it with data?
  - How do we optimize the use of the link bandwidth?

Reliable delivery

- We want the protocol to implement “reliable” frame delivery (as opposed to “best effort” delivery)
- By “reliable”, we mean that frames are:
  - All delivered
  - In order
  - Without bit errors

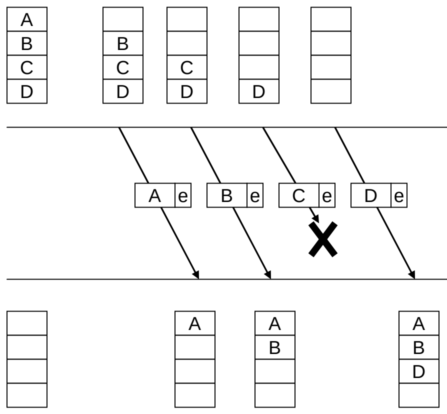
Flow Control Algorithms

- Flow control is essential for ensuring that data is successfully delivered (one part of reliable data delivery)
- Flow control is the set of rules that ensure that receivers aren’t overwhelmed with data (that is, that their receive buffers do not overflow)
- LLC protocol algorithms are often called “flow control algorithms”

The simplest possible algorithm (Simplex)

- Sender:
  - As data is passed from the layer above, pack it into frames and send it.
- Receiver:
  - When a frame is received, unpack it and pass the data to the layer above.

Simplex Algorithm Operation



In Simplex, the sender doesn't know whether or not the receiver has actually received a frame. This leads to out-of-order data.

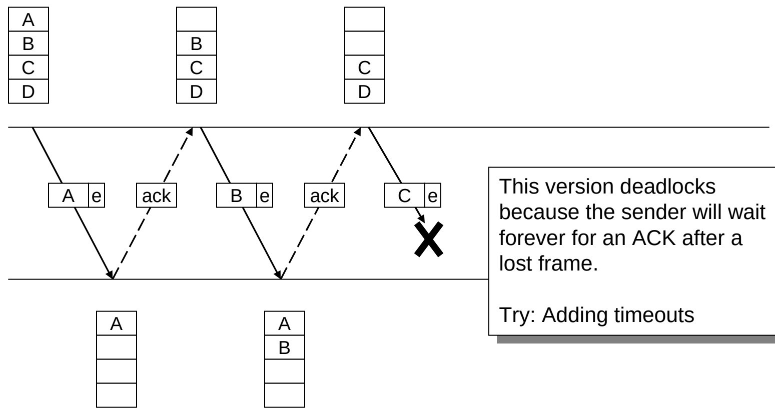
Try: Acknowledging each frame

Simplex + Acknowledgements

The idea:

When the receiver receives a frame, it sends an acknowledgement ("ACK") frame back so that the sender knows the frame was received. The sender will wait until the previous frame is ACKed before sending the next one.

Simplex + Acknowledgements:
Operation



This version deadlocks because the sender will wait forever for an ACK after a lost frame.

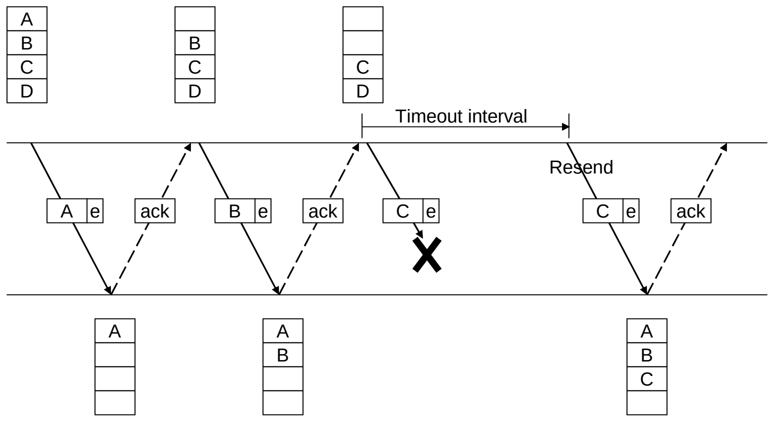
Try: Adding timeouts

Simplex + ACKs + timeouts

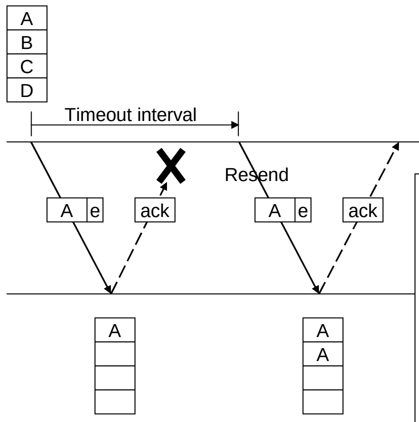
The Idea:

When the sender sends a frame, it starts a timer. If the timer times out before the ACK is received, the sender will re-send the last frame.

Simplex + ACKs + timeouts:
Operation (1)



Simplex + ACKs + timeouts:
Operation (2)



This algorithm gets confused when an ACK is lost. When the sender doesn't get an ACK for the last frame sent, it times out and resends that frame. The receiver does not recognize that it has gotten two copies of the same frame.

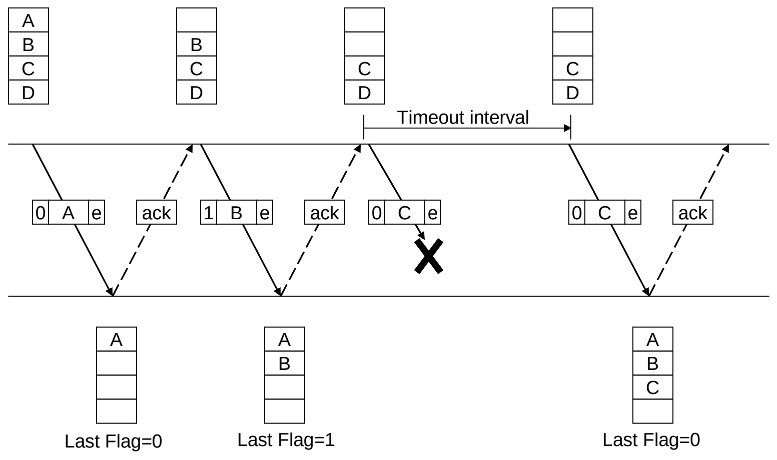
Try: An indicator to show when a frame is repeated

Stop-and-wait Automatic Repeat Request (stop-and-wait ARQ)

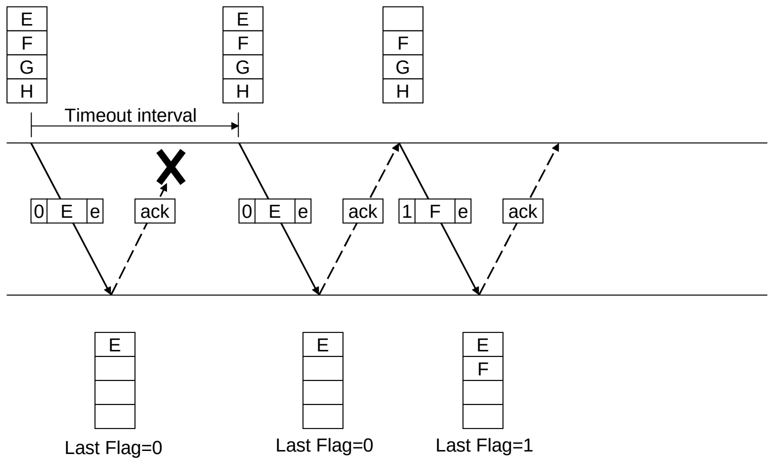
The Idea:

The receiver needs to be able to determine when it has received the same frame twice in a row. We'll add a flag to the frame that alternates between "1" and "0". This way, if the receiver gets two frames in a row that have the same flag, it will know that this is two copies of the same frame and can ignore one of them.

Stop-and-Wait ARQ: Operation (1)



Stop-and-Wait ARQ: Operation (2)



Success!!

We now have a protocol, Stop-and-wait ARQ, that will reliably transfer a frame across the link.

Now, what about efficiency?

Success!! (….sort of)

Say we use SAW ARQ on a 100Mbps link with RTT = 1 msec. Frames are 1000 bits long.

In the best case, when we send a frame, we will wait a little more than the RTT for the ACK. If there are errors, we'll wait longer still.

So in the best case, we are sending about 1 frame per RTT, or 1000 bits per msec.

This works out to an effective bandwidth of $10^3/10^{-3}$ = 1 Mbps.

So we are using just 1% of the capacity of the link.

Surely we can do better than this….

The Go-Back-N Algorithm

The efficiency problem with S-a-W ARQ is due to the link being idle while we wait for an ACK.

The Idea:

While we are waiting for the ACK for a frame, send out the next frames up to a total of "N". This will fill the idle time on the link.

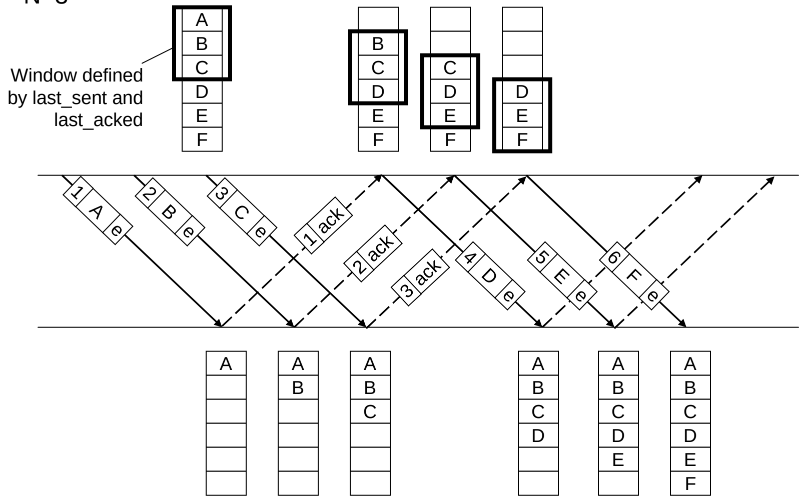
If we receive no ACK for a frame, we will re-send that frame and the N-1 after it. We must number the frames so that the receiver will know which frame it is getting and so that it can specify which frame it is ACKing.

The University Of Alabama in Huntsville Computer Science

Go-Back-N
Operation (1)

cs570

N=3

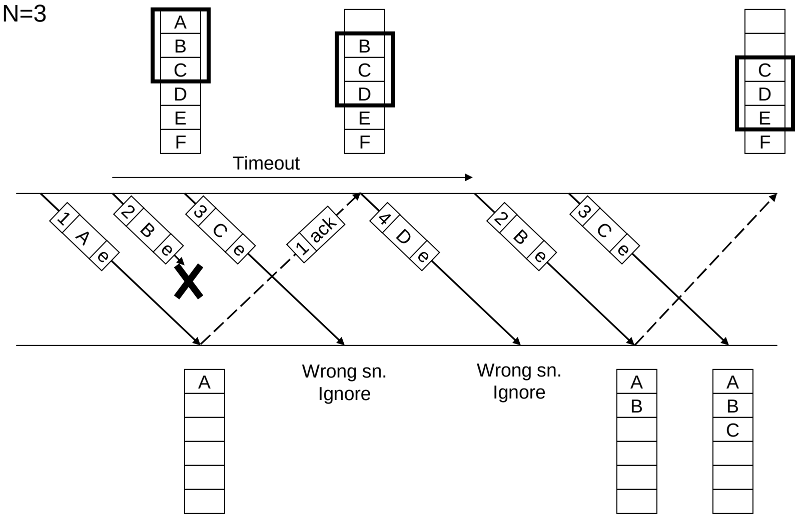


The University Of Alabama in Huntsville Computer Science

Go-Back-N
Operation (2)

cs570

N=3



Better Efficiency, but still a problem

To maximize efficiency, we want to set N large enough so that the link is filled while we're waiting for the ACK (RTT+)

As the length of the link grows, RTT increases and we increase N to maintain high efficiency.

But the higher N grows, the more frames we have to re-send after an error. For an error-prone link, this can cancel any efficiency gains.

Example:
100Mbps link with RTT = 10 msec, Frames1000 bits long.
Each frame takes 1000 bits / 100 Mbps = 10 usec to send.
For max efficiency, set N >= 10msec / 10usec = 1000.

Assume we send 10^6 frames. x% have errors the first time they are sent.

If x=0 it takes 10 sec to send successfully.
If x=0.1%, 10^3 frames will be in error,
so 10^3 x 1000 = 10^6 frames will be re-sent for a total of 2 x 106 frames.
so it will take 20 seconds to send successfully.

The Sliding Window Algorithm

The Idea:

When an error is detected, only the frame in error is re-sent.

(Note that this means the receiver must be able to receive frames out-of-order and order them correctly -- a simple FIFO queue doesn't work).

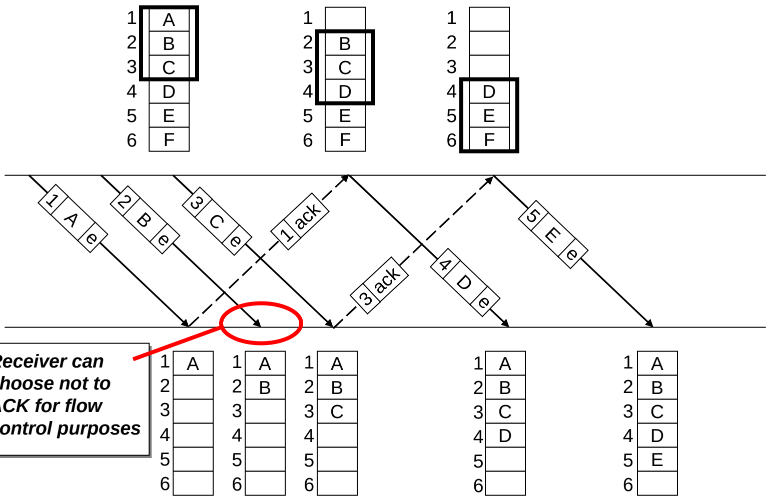
Detail:

– Implicit ACKs: If the receiver ACKs receiving frame F, that means F and all preceding frames are ACKed
– The receiver can control data flow by withholding ACKs. This is usually implemented as a "receive window size" that can be different from the "send window size"

Sliding Window Algorithm
Operation (1)

cs570

Send Window Size=3

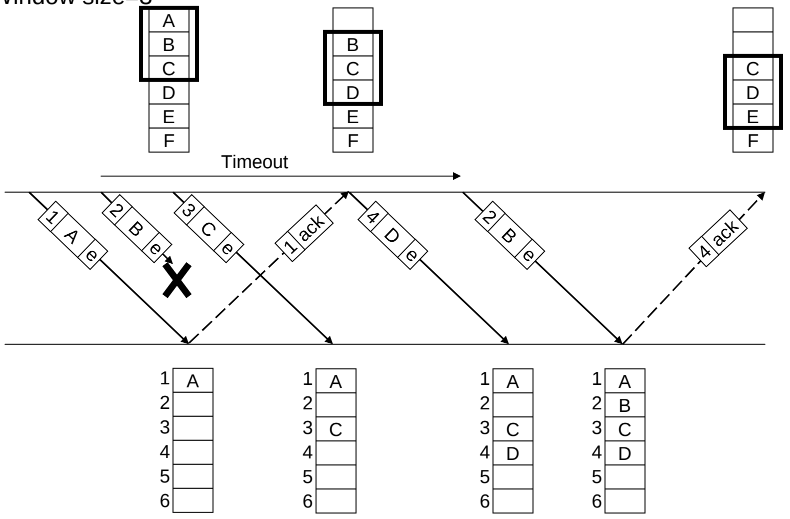


Receiver can choose not to ACK for flow control purposes

Sliding Window Algorithm
Operation (2)

cs570

Window size=3



Success!! (…Really!)

The Sliding Window Algorithm provides reliable data transfer with excellent efficiency

We'll see it again in TCP

Something to notice about Sequence Numbers

- Assume frame sequence numbers are encoded with n bits
- Then they will roll over every 2^n frames
- We can't afford to ever have two frames "in flight" with the same sequence number

- The implications are far-reaching:
  - Send window size can never be greater than 2^n-1
  - Send and receive buffers must hold at least 2^n-1 frames
  - As network speeds increase, we can can have more and more frames in flight -- the 2^n-1 limit becomes more important