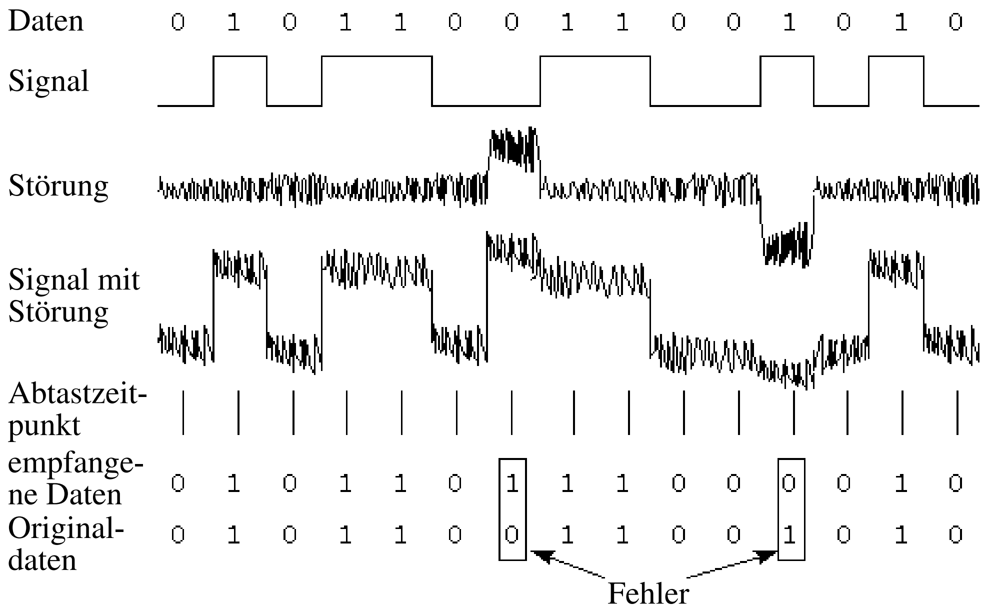
Übertragungsfehler

- Bitfehler durch verrauschten Kanal



- Bitfehler durch fehlerhafte Synchronisation

Übertragungsfehler (2)

- Ursachen für Fehler bei der Datenübertragung
  - thermische Elektronenbewegungen in Halbleitern oder Leitungen
  - elektromagnetische Einstrahlungen (Motoren, Zündanlagen, Blitze)
  - radioaktive Einstrahlungen (z.B. bei Zwischenstationen)
  - schadhafte Geräte (z.B. Versorgungsspannung), Leitungen (Kontakte)
  - zu hohe Leitungsdämpfung; sporadisch fehlerhafte Bitsynchronisation
  - Zugriffskollisionen in Ethernet-LANs
  - Pufferüberlauf beim Empfänger oder einer Zwischenstation
  - unvollständige Nachrichten bei absichtlichem Sendeabbruch
- Oft sind dann mehrere Bits nacheinander falsch
  - Fehlerbündel ("bursts") --> Bitfehler also oft nicht statistisch unabhängig voneinander!
  - einfache Paritätsprüfung versagt dann!
- Fehlerkorrektur bzw. -erkennung ist notwendig
  - selbst bei einer Fehlerrate von "nur" 10^{-7} würde bei 10Mb/s (z.B. Ethernet) jede Sekunde ein Bitfehler auftreten (nicht akzeptabel!)

- Fehlererkennung
  - einfach: Paritätsbit (z.B. "Byteparität"; geht mit XOR in Hardware effizient)
  - es gibt leistungsfähigere Verfahren (z.B. CRC-Codes)
  - typischerweise Datenblock erneut senden bei erkanntem Fehler (wenn Datenblock "beschädigt" wurde oder vermutlich verloren ging)
- Fehlerkorrigierende Codes (z.B. bei Satelliten)
  - "Forward Error Control" (FEC): Empfänger korrigiert verfälschte Bits
  - viel Redundanz notwendig (Hammingdistanz); Probleme bei Fehlerbursts

Datensicherung und Quittungen

- Wesentliche Aufgabe der Ebene 2 (Internet: Ebene 4) ist die *sichere Datenübertragung* und die *Flusssteuerung*

Datenverfälschung und Datenverlust der Ebene 1 ausgleichen

Sendegeschwindigkeit an Geschwindigkeit des Empfängers anpassen

- Daten*verfälschung* i.a. auf Daten*verlust* zurückgeführt
  - Empfänger “vernichtet” Datenblöcke mit erkannten Bitfehlern

- Hilfsmittel für beide Aufgaben sind i.w. *Bestätigungsnachrichten* und *Zeitüberwachungen* (timeouts)
  - positive Bestätigungen (“ACK”)
  - negative Bestätigungen (“NAK”)

  } also zumindest Halbduplex nötig

  - timeout tritt typischerweise bei Ausbleiben eines ACK ein

- Daher beide Aufgaben oft in *einem* Protokoll “verwoben”

- Bei Duplexbetrieb kann ein ACK ggf. zusammen mit einer Nachricht in Gegenrichtung verschickt werden
  - “Huckepack”-Prinzip (“piggybacking”)
  - ACKs oft nur einige Bits lang (z.B. Sequenznummer des letzten erhalten Blocks); eigene Nachricht dafür wäre relativ aufwendig
  - ggf. vor dem Versenden eines ACK kurze Zeit warten, ob dieses nicht huckepack mit der nächsten Nachricht übermittelt werden kann

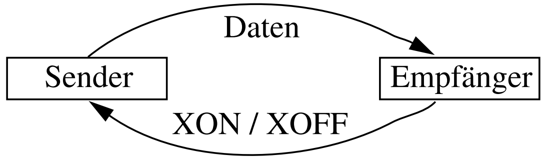
- Erneutes Versenden eines Blocks bei Ausbleiben eines ACK (oder Erhalt eines NAK) wird als “Automatic Repeat Request” (ARQ) bezeichnet

Flusssteuerung

- Aufgabe: Empfänger vor einem zu grossen Zufluss von Paketen eines Senders schützen

- Angesiedelt typischerweise auf der Sicherungsschicht
  - ggf. aber auch (zusätzlich) auf höheren Schichten

- Einfachste Methode: “Halt”- und “Weiter”-Meldung vom Empfänger zum Sender



  - Beispiel: XON/XOFF-Protokoll (nur bei Vollduplex verwendbar!):
    XON ist ASCII-Zeichen 017 (oktal): DC1 (“Device Control 1”);
    XOFF ist ASCII-Zeichen 019 (oktal): DC3 (“Device Control 3”)
    (auf ASCII-Tastaturen: XON/XOFF mit “control Q” bzw. “control S”)
  - muss rechtzeitig eintreffen (beachte Bandbreite-Delay-Produkt!)

- Implizite Flusssteuerung, z.B.:
  - Zurückhalten der Quittung (ACK)
  - Empfänger räumt dem Sender einen mehrere Transfereinheiten umfassenden Sendekredit ein; bei Erschöpfung (ohne neue Kreditgewährung) stoppt der Sender
  - Beachte: Kreditmethode erfordert erhöhte Fehlerkontrolle (z.B. bzgl. Verlust einer Kreditgewährung)

- Ratenbasierte Flusssteuerung
  - Empfänger erlaubt dem Sender, eine bestimmte Menge an Daten innerhalb eines Zeitintervalls zu senden
  - bei vorzeitiger Ausschöpfung der Datenmenge muss der Sender bis zum nächsten Zeitintervall warten

Netzüberlastung und Laststeuerung

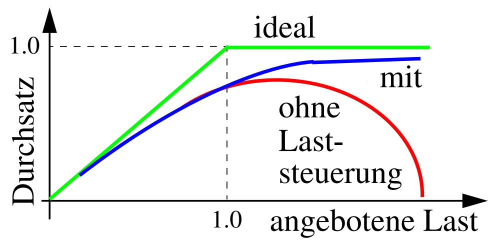
network congestion   congestion control

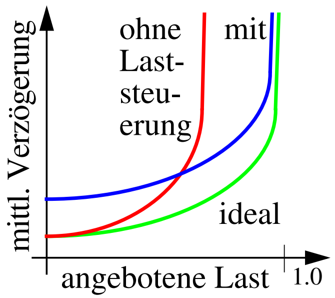
- Bei Netzüberlastung: grosse Verzögerungen, sinkender Durchsatz; Datenverlust durch Pufferüberläufe

- Gründe für (lokale oder globale) Netzüberlastung:
  - zu starker Zufluss von Datenpaketen
  - Umkonfiguration aufgrund von Teilausfällen des Netzes
  - Zunahme transienter Störungen in Netzteilen

- Zweck von Laststeuerung (bzw. “Staukontrolle”):
  - “vernünftiges” Netzverhalten bei Hochlast-/ Überlastbetrieb
  - Erhaltung bzw. Optimierung der Durchsatzleistung (Last so begrenzen, dass keine Reduktion des Durchsatzes erfolgt)
  - Verhinderung von Pufferüberläufen aufgrund von Überlastung

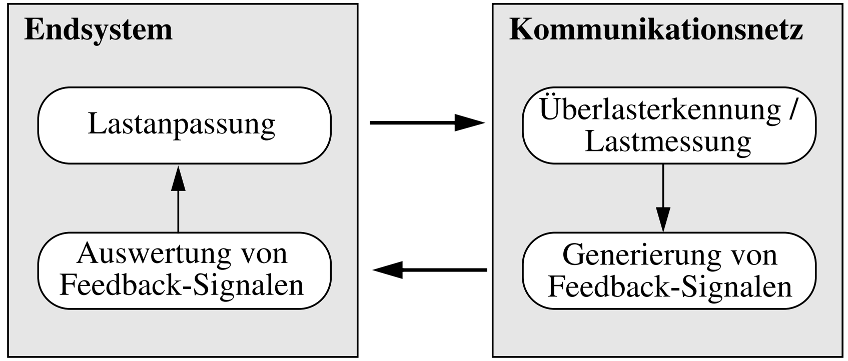
- Ohne Laststeuerung kann es zu einem Kollaps kommen
  - weggeworfene Pakete (Pufferüberlauf!) werden wiederholt und vergrössern noch die Netzlast





  - Laststeuerung hat einen Overhead (spürbar im Niedriglastbereich); darf aber nicht zur Verschlimmerung im Hochlastbereich beitragen!

Einige Methoden der Laststeuerung



- “Choke-Paket” mit “stop” oder “langsamer” von einem überlasteten Netzknoten an seine Quellen
- Informieren über den Netzzustand durch kontinuierliches Versenden von Paketen, die die Ende-zu-Ende-Verzögerung messen
- Lasteinspeisung zurückfahren, wenn Verlustrate steigt, Quittungen ausbleiben etc. und bei Besserung allmählich wieder hochfahren

- Anforderungen und Probleme z.B.:
  - schnelle Reaktion auf plötzliche Änderung der Lastsituation
  - Überreaktion vermeiden
  - bei Überlast Durchsatz stets nahe 100% halten
  - Fairness (keine wesentliche Benachteiligung einzelner Teilnehmer)

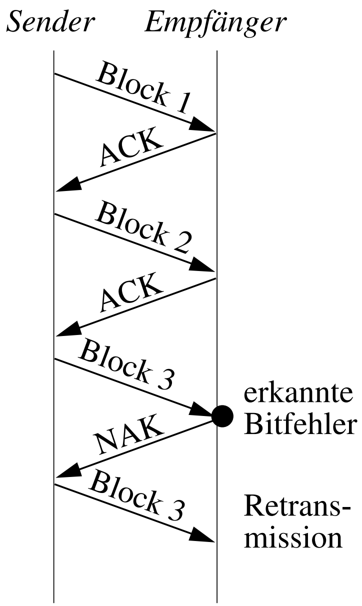
- Reservierung von Ressourcen bei Verbindungsaufbau?
  - Problem: verschwendet ggf. ungenutzte Kapazität
  - für verbindungslose Kommunikation (z.B. Internet-IP) eher ungeeignet

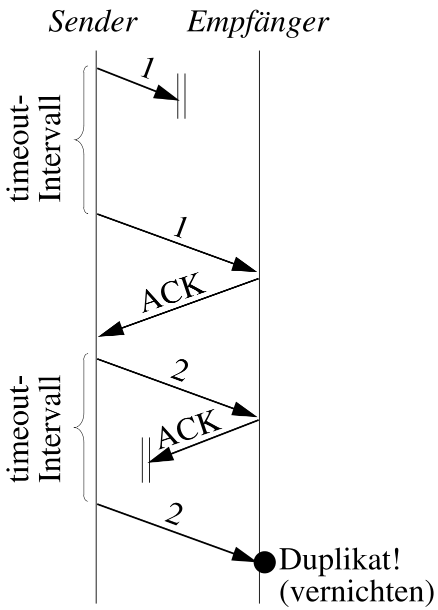
Stop-and-Wait-Protokoll

(gelegentlich auch als “Send-and-Wait” bezeichnet)

- Einfachstes Protokoll zur Datensicherung, Flusssteuerung und Reihenfolgegarantie

ACK erst senden, wenn wieder Puffer verfügbar

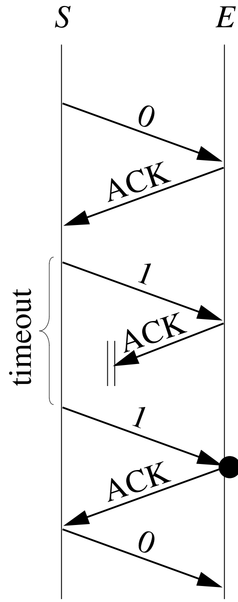




- Senderseitige timeouts, damit es bei Verlust von Blöcken oder ACKs nicht zu Blockaden kommt
  - damit dient NAK lediglich der Beschleunigung
  - ohne NAK: PAR-Protokoll (“Positive Acknowledge and Retransmit”)
  - timeout-Intervalle “richtig” wählen
    - unnötiges Warten <--> unnötige Wiederholung
    - schwierig bei grosser Varianz bei den Übertragungszeiten
    - ggf. adaptiv an Übertragungszeit, Fehlerrate etc. anpassen

- Frage: Sind “zu langsame” ACKs eigentlich problematisch?

Alternating-Bit



- Zum Erkennen von Duplikaten werden Datenblöcke i.a. numeriert
  - bei Retransmission Sequenznummer beibehalten
- Bei Stop-and-Wait genügen Sequenznummern mod 2 (“alternating bit”)!
  - das gilt bei später betrachteten Protokollen i.a. nicht!

Dies muss ein Duplikat des vorherigen Blocks sein, denn

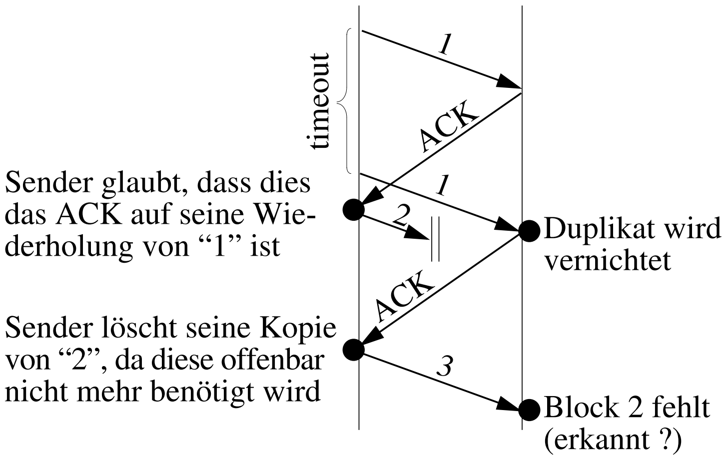
- bei dieser “1” kann es sich nicht um eine “1” aus der Zukunft handeln (weil der Sender vor dem Senden anderer Blöcke das ACK abwartet)
- es kann sich auch nicht um einen mit “1” markierten früheren Block handeln (weil alle früher versendeten Blöcke bereits korrekt empfangen wurden)

- Wie überzeugend ist eigentlich eine solche Argumentation?
  - zumindest bei verwickelteren Abläufen kann man sich leicht täuschen
  - formalere Modelle zur Spezifikation und Verifikation von Protokollen scheinen daher notwendig zu sein

- Mit dem ACK schickt man i.a. die Sequenznummer des bestätigten Blocks mit
  - hier also ACK(0) bzw. ACK(1)
  - aber nützt das etwas oder kann man darauf verzichten?

Ein Protokollfehler?

- Wir betrachten nochmals das *Stop-and-Wait*-Protokoll:

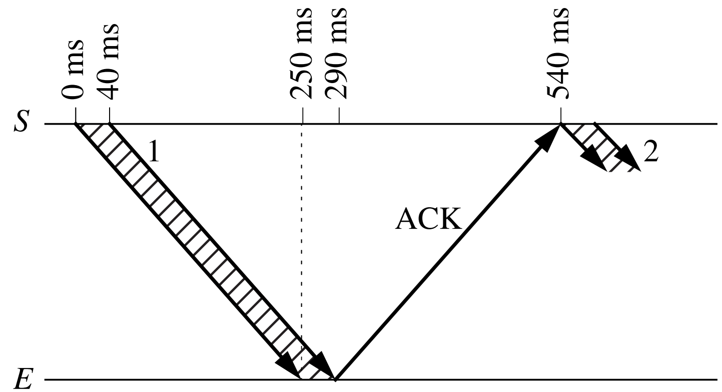


- Hier war das erste ACK so langsam, dass es erst *nach* dem timeout des Senders ankam

- Es ergeben sich einige Fragen...
    - Ist das Protokoll tatsächlich falsch? Lässt es sich einfach "reparieren"?
    - Hätten wir alle das Problem vor der Implementierung erkannt?
    - In verwickelteren Fällen auch?
    - War die Argumentation beim Alternating-Bit vielleicht doch nicht ganz wasserdicht?
    - Ist das der einzige (bzw. letzte) "bug" des Protokolls?
    - Denkübung: Wie kann man das Problem möglichst einfach lösen?
    - Helfen dabei Sequenznummern in den Acks? (Beachte auch Alternating-Bit und sehr langsame ACKs!)
    - Wie kann man sich überhaupt jemals sicher sein?

Kanalausnutzung von Stop-and-Wait

- Es kann stets höchstens ein Block "unterwegs" sein, z.B.:
    - Satellitenlink mit 500 ms Round-Trip-Delay
    - Übertragungsrate 50 kb/s
    - Datenblöcke von 2 kb



- Bandbreite-Delay-Produkt = 50 kb/s $\times$ 0.5 s = <u>25 kb</u>

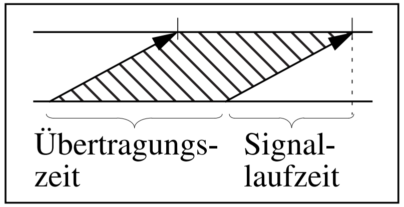
- Speicherkapazität des Kanals mit 2 kb also schlecht genutzt

- Da ACKs vernachlässigbar kurz sind, ergibt sich genauer eine Ausnutzung ("Effizienz") von nur 40/540 = 7.4%

Effizienz von Stop-and-Wait

- Betrachte einen Parameter a:

$$a = \frac{\text{Signallaufzeit}}{\text{Übertragungszeit}}$$



- Es folgt damit:

$$a = \frac{\text{Bandbreite x Signallaufzeit}}{\text{Blockgrösse}}$$

bei konstanter Blockgrösse ist dies also proportional zum "Bandbreite-Delay-Produkt"!

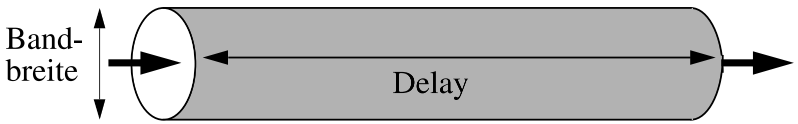
- Für die Effizienz E ("Durchsatz / Kapazität") gilt:

$$E = \frac{1}{1 + 2a}$$

(Denkübung: wieso? Zur Plausibilität setze a=1: Block füllt die Pipe; je eine Zeiteinheit bis Ende der Übertragung, Ankunft des letzten Bits, Ankunft Ack.)

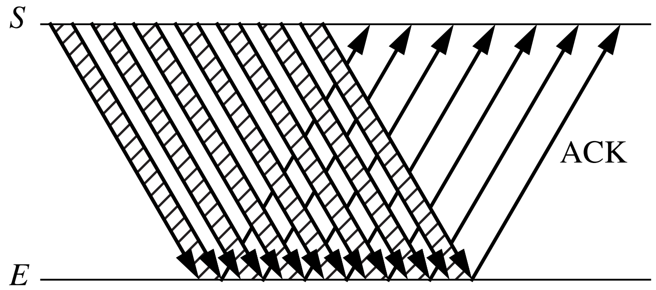
- Beispiel 1 (von vorhin): a = 50 kb/s x 250 ms / 2 kb = 6.25 --> **E = 7.4%**
- Beispiel 2: LAN, Blockgrösse 1000 Bit, Übertragungsrate 10 Mb/s, Entfernung 100 - 10000 m --> **E ≈ 0.5 - 0.99**
- Beispiel 3: Modem 28.8 kb/s, Blockgrösse 1000 Bit
  1000 m --> **E ≈ 100%**
  5000 km --> **E ≈ 40%**
- Beispiel 4: ATM (Blockgrösse 424 Bits, Glasfaser mit V=2 x 10^8 m/s, 155 Mb/s), 1000 km --> **E = 0.027%**

- Wie kann man das verbessern?
  - Motto: Keep the pipe full!



Pipelining

- Auch als "continuous ARQ" bezeichnet
- Idee: Sender setzt mehrere Blöcke ab, ehe er wegen evtl. ausstehenden ACKs blockiert



- ACKs müssen den Blöcken eindeutig zuordenbar sein
- Sender muss mehrere Blöcke aufheben, um ggf. einen Block erneut senden zu können
- Sender hat also quasi einen "Kredit" zum Senden von Blöcken
- wie hoch sollte man den Kredit wählen?
- Timeout-Verwaltung und Retransmissionsverwaltung komplexer

==> Sliding-Window-Prinzip

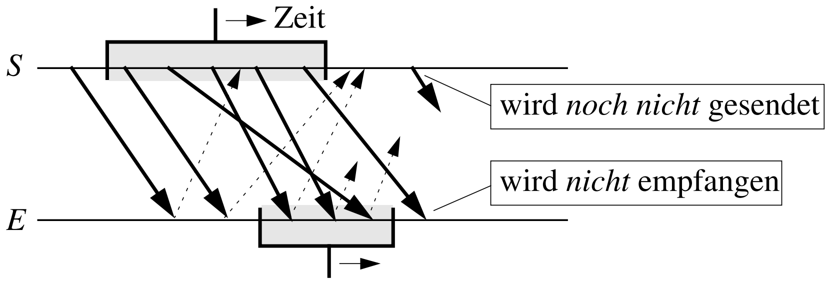
Sliding-Window-Protokoll

"Schiebefenster"

- Protokoll (der Ebene 2 oder höher) für
  - sichere Datenübertragung mittels ARQ
  - Flusssteuerung
  - Reihenfolgegarantie (FIFO-Übertragung)

- Eigentlich eine Familie von Protokollen
  - verschiedene spezifische Varianten

- "Verbesserte" Version von "Stop-and-Wait"
  - also Wiederholung nach timeout für ACK
  - nutzt aber Pipelining zur besseren Auslastung des Kanals

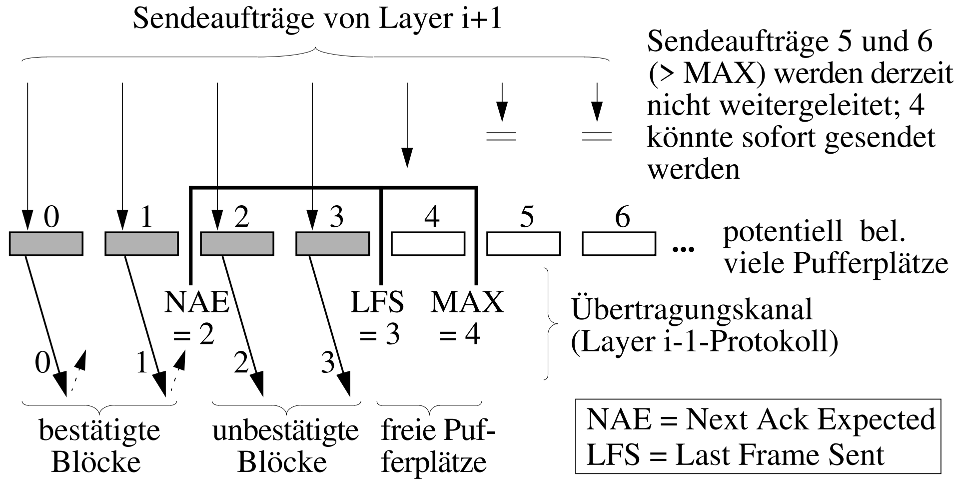


- Sendefenster (Grösse variabel aber begrenzt)
  - Maximale Fenstergrösse entspricht dem "Kredit" zum Senden
  - Aktuelle Fenstergrösse entspricht Zahl ggw. unbestätigter Blöcke

- Empfangsfenster (Grösse i.a. fest)
  - Fenstergrösse bestimmt Zahl der Blöcke, die ausserhalb der Reihe ankommen können
  - Grösse kann verschieden von der Maximalgrösse des Sendefensters sein

Sliding-Window: Senderseite

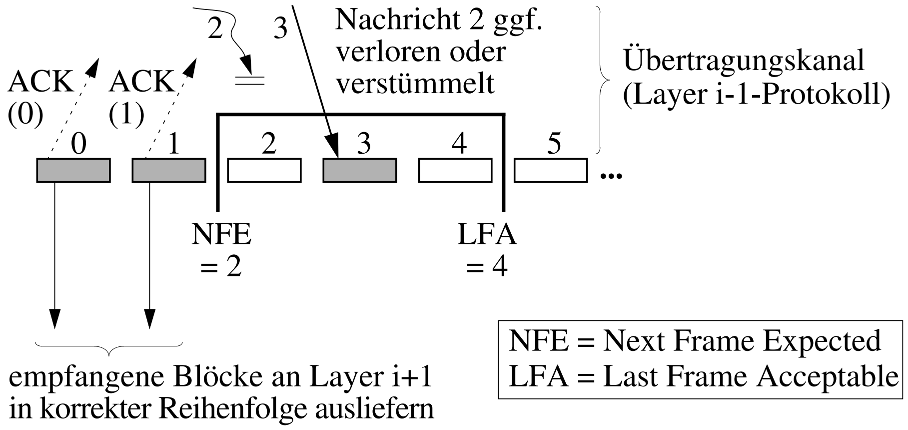
- Alle Nachrichten (und ACKs) seien numeriert: 0, 1, 2,...



- Aktuelle Fenstergrösse = LFS-NAE+1 (hier: 2)
- Maximale Fenstergrösse = MAX-NAE+1 (hier: 3)
- Blöcke innerhalb des Fensters müssen aufbewahrt werden (müssen ggf. wiederholt gesendet werden)
- Senden eines Blockes --> LFS++ (LFS stets ≤ MAX)
- Wenn ein ACK für NAE ankommt, kann das Fenster verschoben werden (NAE++ und MAX++)
  - dadurch können ggf. wieder Pufferplätze frei werden (falls LFS = MAX war) bzw. wartende Blöcke versendet werden
  - es kann aber auch ein anderes ACK(k) mit k ≤ LFS ankommen
- Oft: ACK(k) impliziert ACK(k-1), ACK(k-2)...
  - kumulative ACKs sparen ggf. einige ACKs ein
  - ggf. mit Versenden von ACK(k) auf ACK(k+1) warten...

Sliding-Window: Empfangsseite

- Empfangsfenster bestimmt Bereich von Sequenznummern, die gegenwärtig akzeptiert werden
  - entsprechend viele Pufferplätze sind bereitzuhalten



- Blöcke ausserhalb dieses Bereiches werden vernichtet
  - Seqnr < NFE --> Duplikat (aber ACK versenden! (wieso?))
  - Seqnr > LFA --> kein Pufferplatz vorhanden
    (kann das passieren, wenn Sendefenster “richtig” dimensioniert ist?)
- Innerhalb des Fensters können Blöcke aber in beliebiger Reihenfolge ankommen
  - oder erst mal ausbleiben (bis der Sender fehlende Blöcke wiederholt)
- Bei Empfang von Block(NFE): zugehöriges ACK senden und Fenster verschieben: NFE++; LFA++
  - ggf. um mehr als 1 verschieben, wenn anschliessende Blöcke schon da
  - bei Empfang eines Blockes > NFE: Soll ein ACK versendet werden?
    (Je nach Variante: nein; ja; ja aber erneut mit ACK(NFE)...)

Fehlerbehandlung bei Sliding-Window

- Block ist verloren oder wurde (z.B. mittels CRC bzw. Prüfsumme) als fehlerhaft erkannt
  - Empfänger könnte zusätzlich auch ein NAK senden, wenn ein Block fehlerhaft ist oder er eine Lücke bei den Sequenznummern entdeckt
- Es gibt zwei prinzipielle Möglichkeiten:

1) “go back n”:

- Falls der Sender einen Block k wiederholt, sendet er auch alle schon versendeten Blöcke k+1, k+2... erneut
- Frage: Wie reagiert der Empfänger?
- Einfach, aber schlechter Durchsatz bei hoher Fehlerrate

2) “selective repeat”:

- Nur fehlerhafter Block wird wiederholt
- Beim Sender angestossen durch timeout, durch NAK oder durch Empfang eines anderen ACK als erwartet (je nach Variante)
- Wenn eine Retransmission eintrifft, entsteht an der Dienstschnittstelle zu Layer i+1 u.U. ein “burst” von angebotenen Blöcken