

Rechnernetze I

SoSe 2024

Roland Wismüller
Universität Siegen
roland.wismueller@uni-siegen.de
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: 13. Februar 2024

Rechnernetze I

SoSe 2024

7 Ende-zu-Ende Protokolle



Inhalt

- ➔ Ports: Adressierung von Prozessen
 - ➔ UDP
 - ➔ TCP (Bytestrom, Paketformat)
 - ➔ Sicherung der Übertragung
 - ➔ Übertragungssicherung in TCP
 - ➔ Überlastkontrolle
 - ➔ TCP Verbindungsauf- und abbau
-
- ➔ Peterson, Kap. 5.1, 5.2.1 – 5.2.4, 5.2.6
 - ➔ CCNA, Kap. 9

Einordnung

➔ **Protokolle der Vermittlungsschicht:**

- ➔ Kommunikation zwischen **Rechnern**
- ➔ Adressierung der Rechner
- ➔ IP: *Best Effort*, d.h. keine Garantien

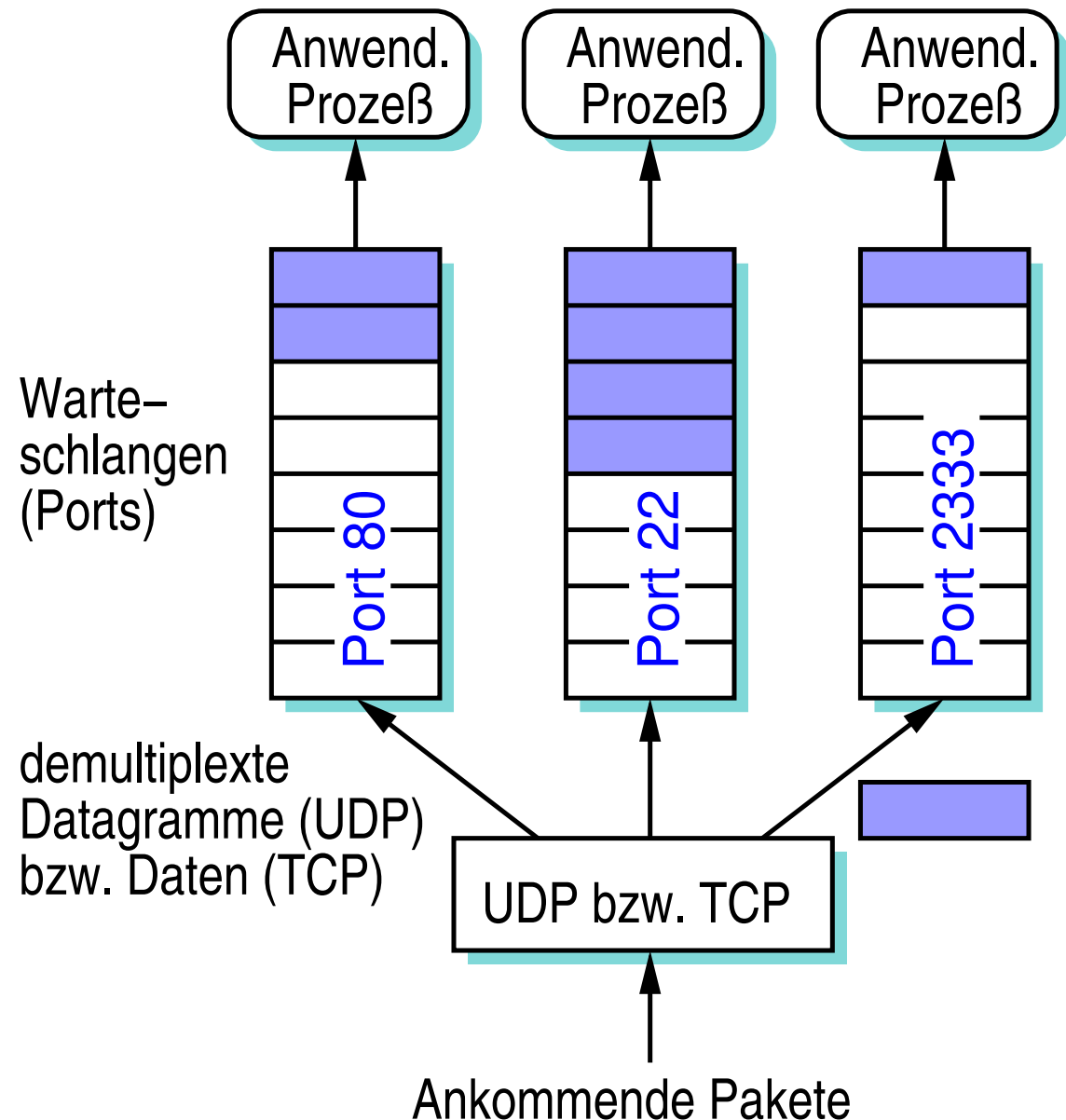
➔ **Protokolle der Transportschicht:**

- ➔ Kommunikation zwischen **Prozessen**
- ➔ Adressierung von Prozessen auf einem Rechner
- ➔ ggf. Garantien: Zustellung, Reihenfolge, ...
 - ➔ Sicherung der Übertragung notwendig
- ➔ zwei Internet-Protokolle: UDP und TCP

- ➔ Wie identifiziert man Prozesse?
 - ➔ **Nicht** durch die Prozeß-ID des Betriebssystems:
 - ➔ systemabhängig, „zufällig“
 - ➔ **Sondern:** indirekte Adresierung über Ports
 - ➔ 16-Bit Nummer
- ➔ Woher weiß ein Prozeß die Port-Nummer des Partners?
 - ➔ „*well known ports*“ (i.d.R. 0...1023) für Systemdienste
 - ➔ z.B.: 80 = Web-Server, 25 = Mail-Server
 - ➔ Analogie: Tel. 112 = Feuerwehr
 - ➔ Server kennt die Port-Nummer des Clients aus UDP- bzw. TCP-Header der Anfrage

Port-Demultiplexing

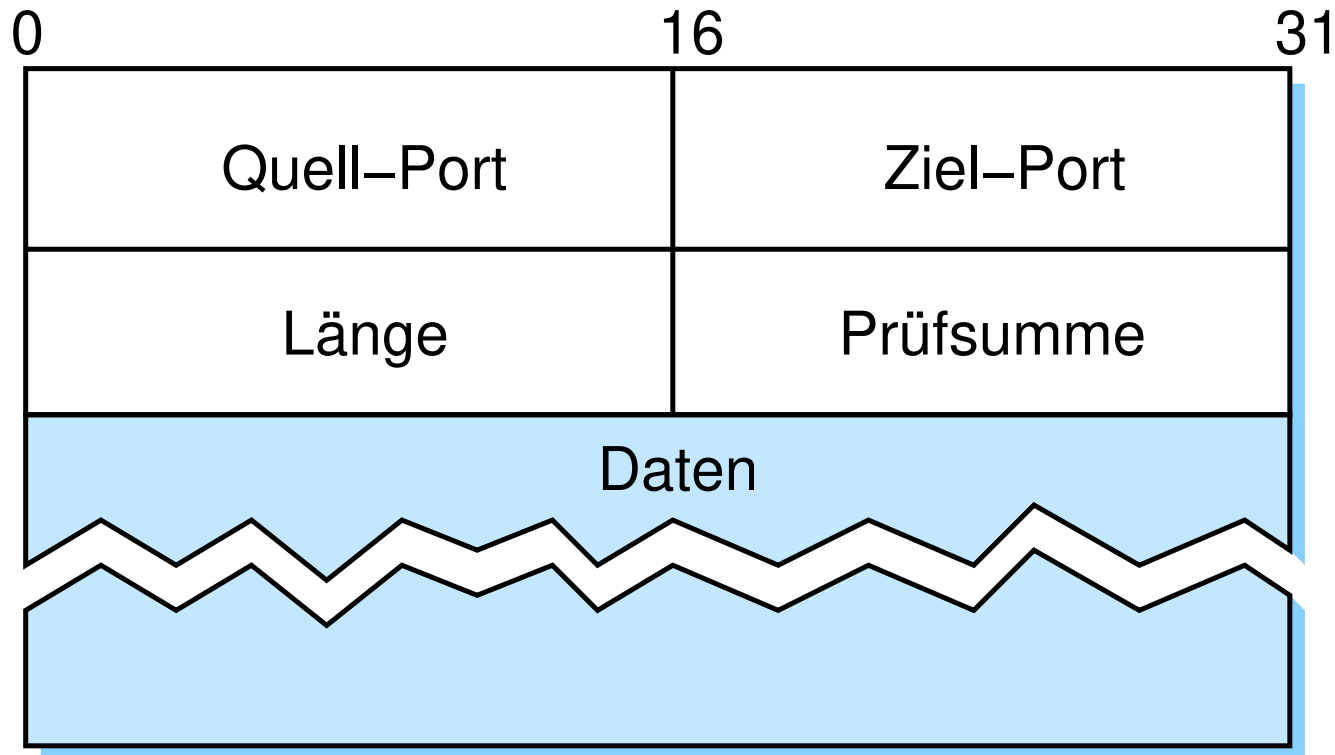
- ➔ Ports sind typischerweise durch Warteschlangen realisiert
- ➔ Bei voller Warteschlange: UDP bzw. TCP verwirft das Paket



UDP: *User Datagram Protocol*

- ➔ Dienstemodell von UDP:
 - ➔ Übertragung von Datagrammen zwischen Prozessen
 - ➔ unzuverlässiger Dienst
- ➔ „Mehrwert“ im Vergleich zu IP:
 - ➔ Kommunikation zwischen Prozessen
 - ➔ ein Prozess wird identifiziert durch das Paar (Host-IP-Adresse, Port-Nummer)
 - ➔ UDP übernimmt das Demultiplexen (☞ 7.1)
 - ➔ d.h. Zustellung an Zielprozess auf dem Zielrechner
 - ➔ Prüfsumme über Header und Nutzdaten

Aufbau eines UDP-Pakets



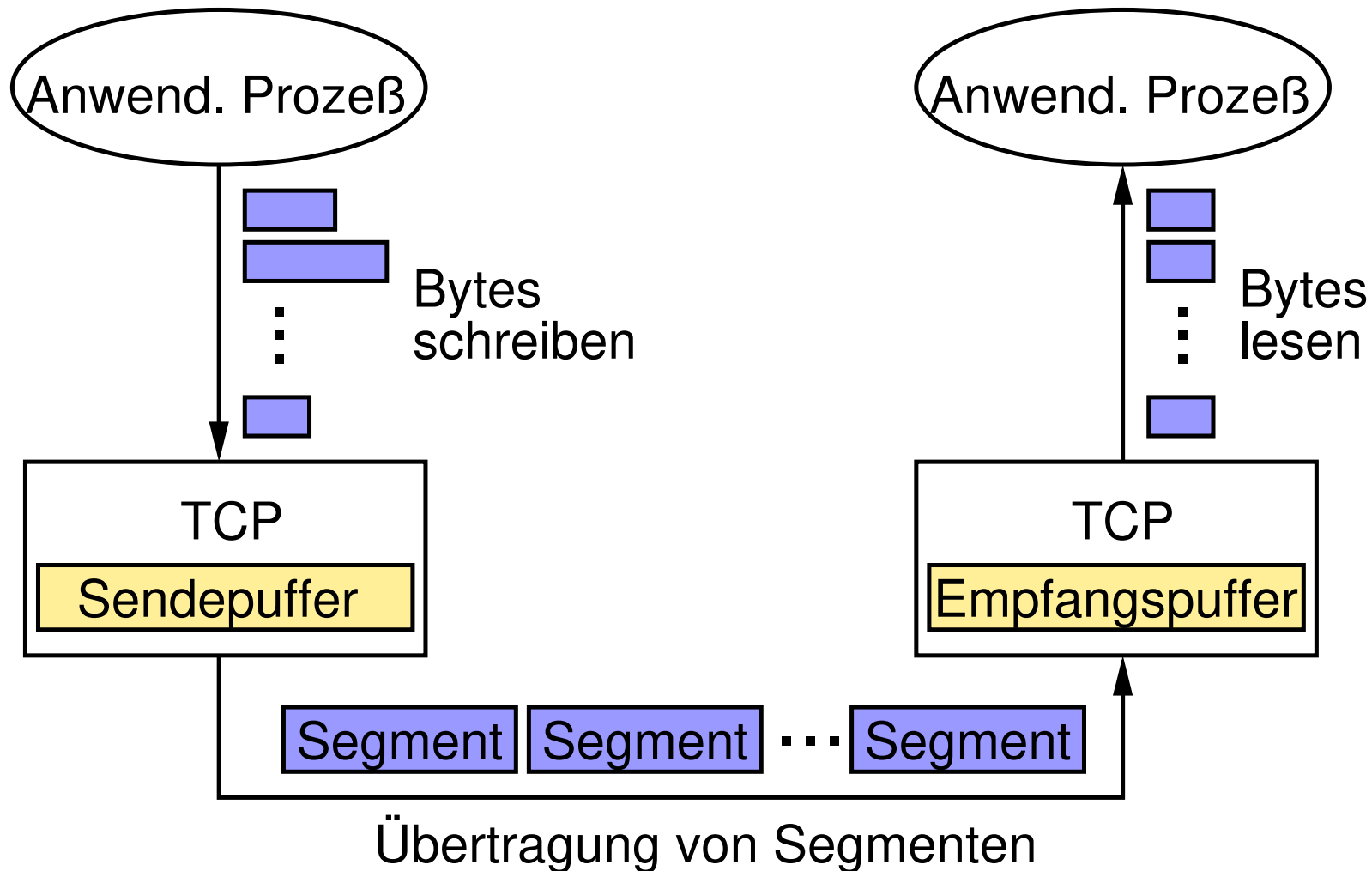
➔ (Das UDP-Paket ist im Nutzdatenteil eines IP-Pakets!)

TCP: *Transmission Control Protocol*

- ➔ Dienstemodell von TCP:
 - ➔ verbindungsorientierte, zuverlässige Übertragung von Datenströmen zwischen Prozessen
- ➔ Meist verwendetes Internet-Protokoll
 - ➔ befreit Anwendungen von Sicherung der Übertragung
- ➔ TCP realisiert:
 - ➔ Port-Demultiplexing (☞ 7.1)
 - ➔ Sicherung der Übertragung, Reihenfolgeerhaltung
 - ➔ Flußkontrolle
 - ➔ Überlastkontrolle

Bytestrom-Übertragung

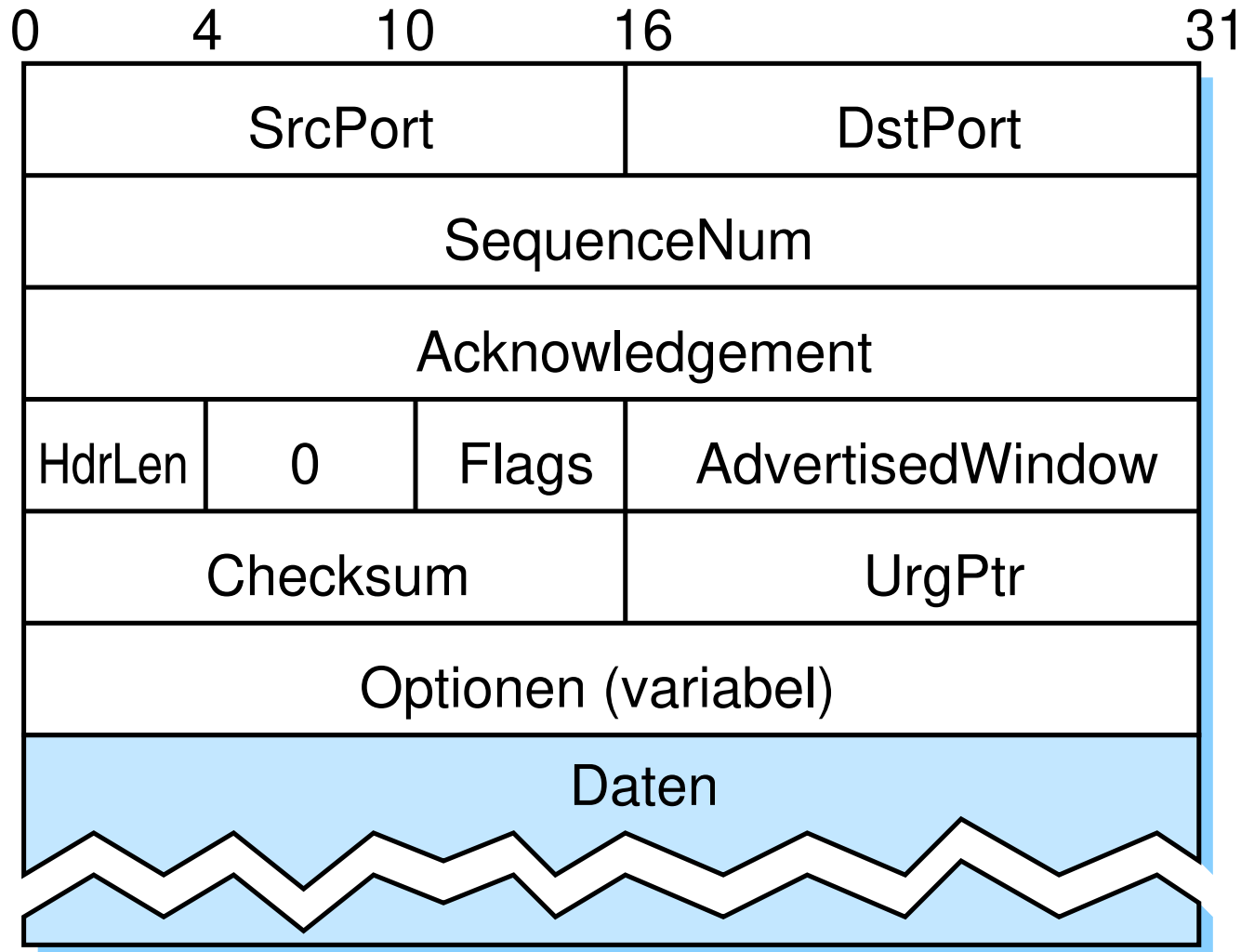
➔ TCP überträgt Daten (Byteströme) segmentweise



Wann wird ein Segment gesendet?

- ➔ Wenn die maximale Segmentgröße erreicht ist
 - ➔ **Maximum Segment Size (MSS)**
 - ➔ i.d.R. an maximale Frame-Größe (**MTU**, **Maximum Transmission Unit**) des lokalen Netzes angepaßt:
 - ➔ $MSS = MTU - \text{Größe(TCP-Header)} - \text{Größe(IP-Header)}$
 - ➔ verhindert, daß das Segment von IP sofort wieder fragmentiert werden muß (☞ 5.4)
- ➔ Wenn der Sender es ausdrücklich fordert
 - ➔ *Push-Operation (Flush)*
- ➔ Nach Ablauf eines periodischen Timers

Aufbau eines TCP Segments



➔ (Das TCP-Segment ist im Nutzdatenteil eines IP-Pakets!)

Aufbau eines TCP Segments ...

- ➔ **SequenceNum, Acknowledgement, AdvertisedWindow:**
 - ➔ für *Sliding-Window*-Algorithmus (siehe später, **7.5**)
- ➔ **Flags:**
 - ➔ SYN Verbindungsaufbau
 - ➔ FIN Verbindungsabbau
 - ➔ ACK **Acknowledgement**-Feld ist gültig
 - ➔ URG Dringende Daten (*out of band data*)
 UrgPtr zeigt Länge der dringenden Daten an
 - ➔ PSH Anwendung hat *Push*-Operation ausgeführt
 - ➔ RST Abbruch der Verbindung (nach Fehler)

Es können mehrere Flags gleichzeitig gesetzt sein

Problem:

- ➔ Bei der Übertragung eines Segments bzw. Frames können Fehler auftreten
 - ➔ Empfänger kann Fehler erkennen, aber i.a. nicht korrigieren
 - ➔ Segmente bzw. Frames können auch ganz verloren gehen
 - ➔ z.B. durch überlasteten Router bzw. Switch
 - ➔ oder bei Verlust der Frame-Synchronisation (👉 **3.5**)
- ➔ Segmente bzw. Frames* müssen deshalb ggf. neu übertragen werden

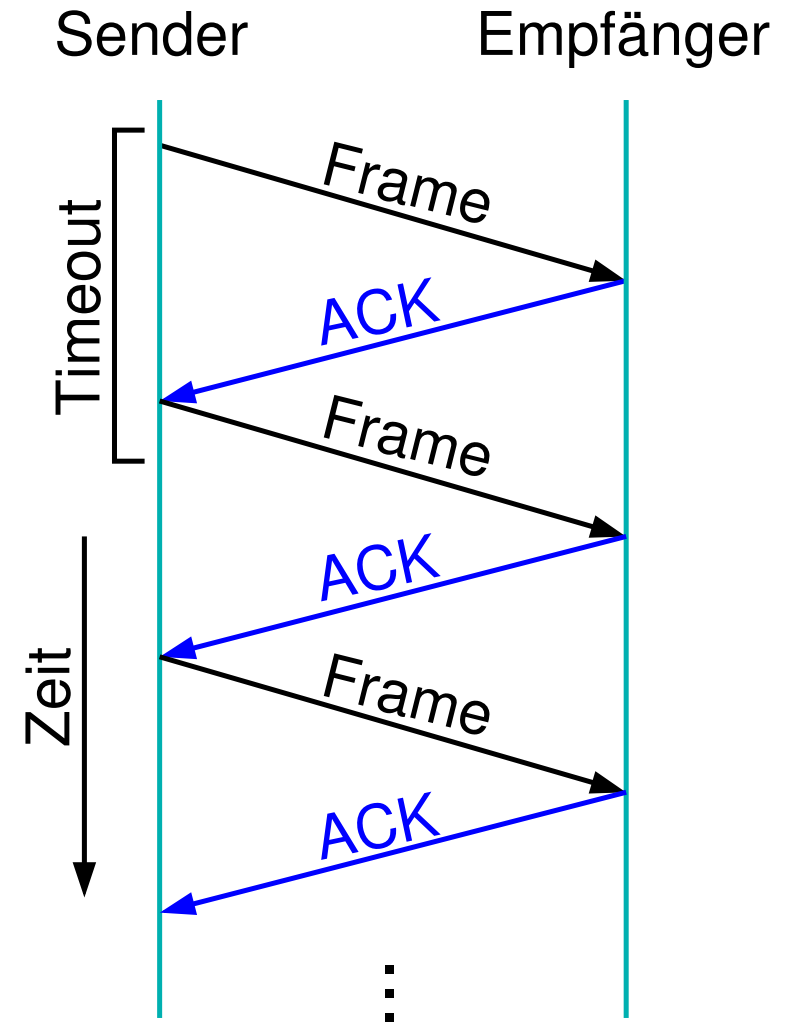
* Zur Vereinfachung wird im Folgenden nur der Begriff „Frame“ verwendet!

Basismechanismen zur Lösung:

- ➔ **Bestätigungen** (*Acknowledgements*, ACK)
 - ➔ spezielle Kontrollinformationen, die an Sender zurückgesandt werden
 - ➔ bei Duplex-Verbindung (wie z.B. bei TCP) auch **Huckepack-verfahren** (*Piggyback*):
 - ➔ Bestätigung wird im Header eines normalen Frames übertragen
- ➔ Senderseitige Zwischenspeicherung unbestätigter Frames
- ➔ **Timeouts**
 - ➔ wenn nach einer bestimmten Zeit kein ACK eintrifft, überträgt der Sender den Frame erneut

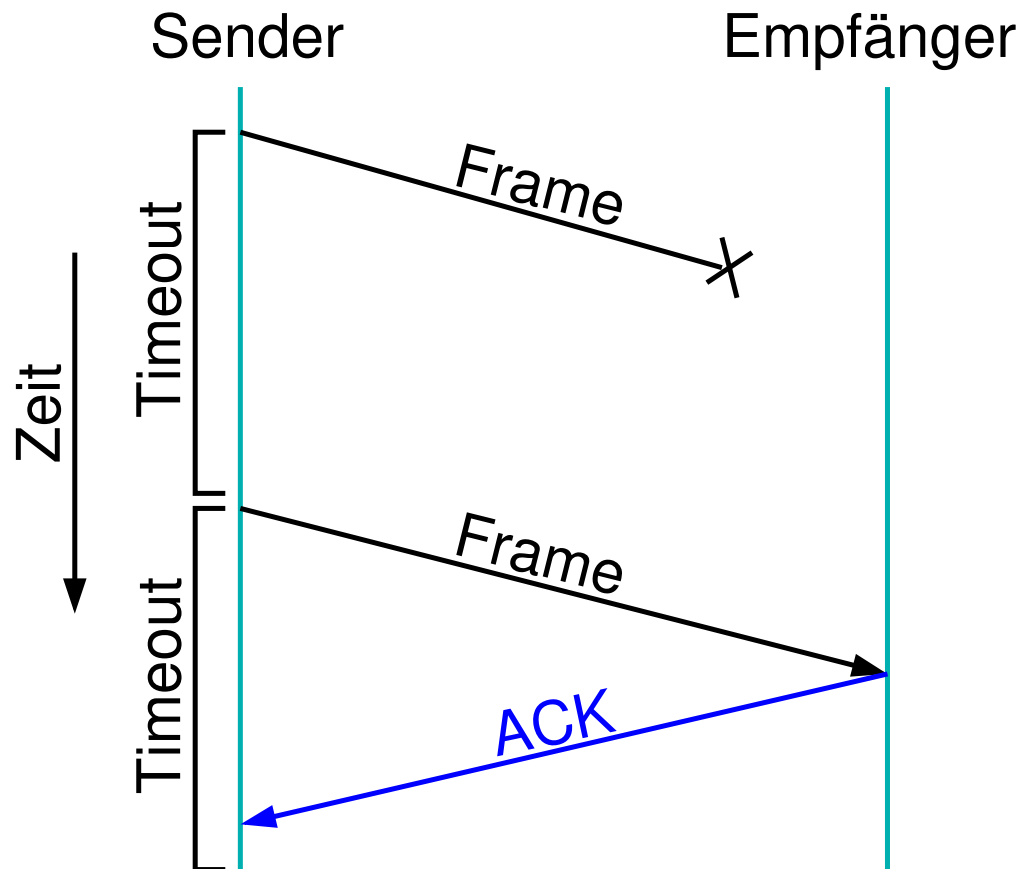
Ablauf bei fehlerfreier Übertragung

- ➔ Sender wartet nach der Übertragung eines Frames, bis ACK eintrifft
- ➔ Erst danach wird der nächste Frame gesendet



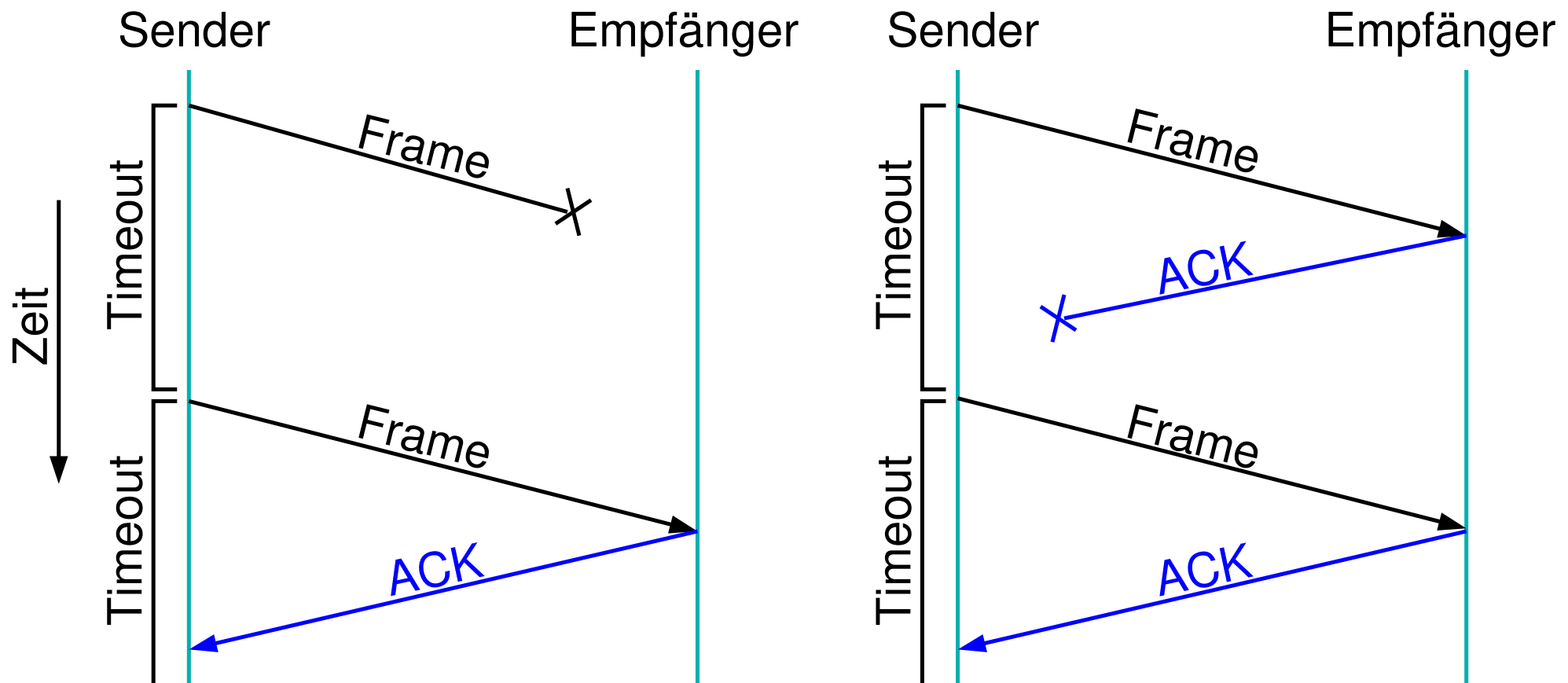
Ablauf bei Übertragungsfehler

- ➔ Falls ACK nicht innerhalb der Timeout-Zeit eintrifft:
 - ➔ Wiederholung des gesendeten Frames



Ablauf bei Übertragungsfehler

- ➔ Falls ACK nicht innerhalb der Timeout-Zeit eintrifft:
- ➔ Wiederholung des gesendeten Frames

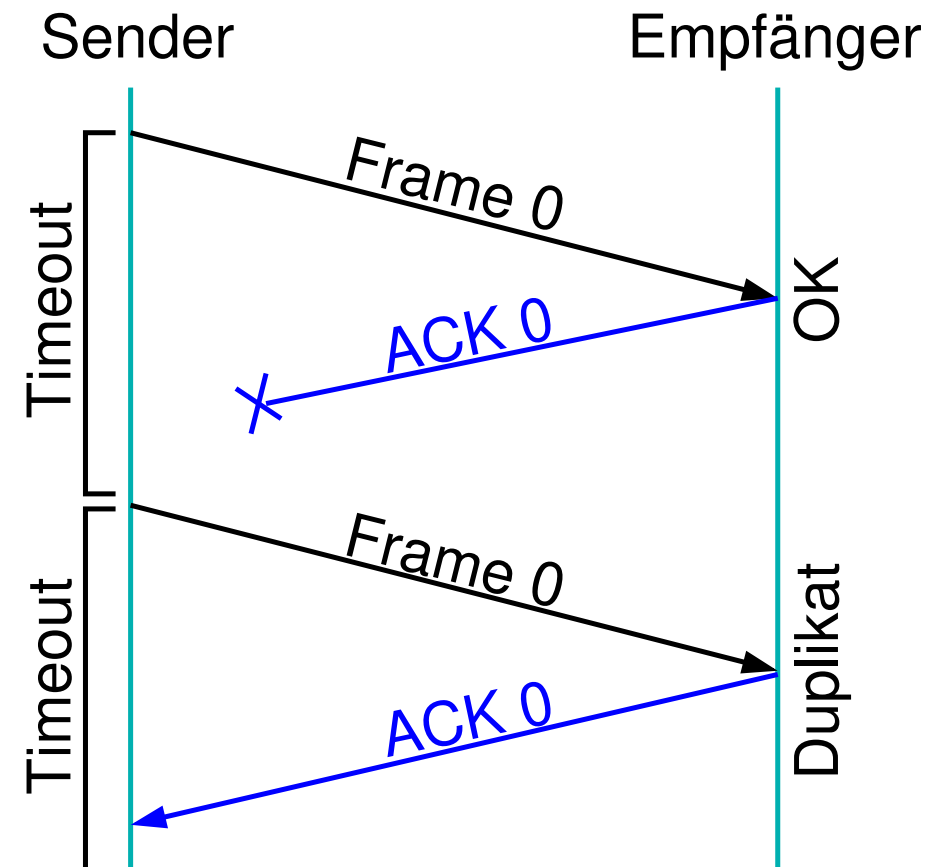


Was passiert, wenn ACK verloren geht oder zu spät eintrifft?

- ➔ Der Empfänger erhält den Frame mehrfach
- ➔ Er muß dies erkennen können!

- ➔ Daher: Frames und ACKs erhalten eine **Sequenznummer**

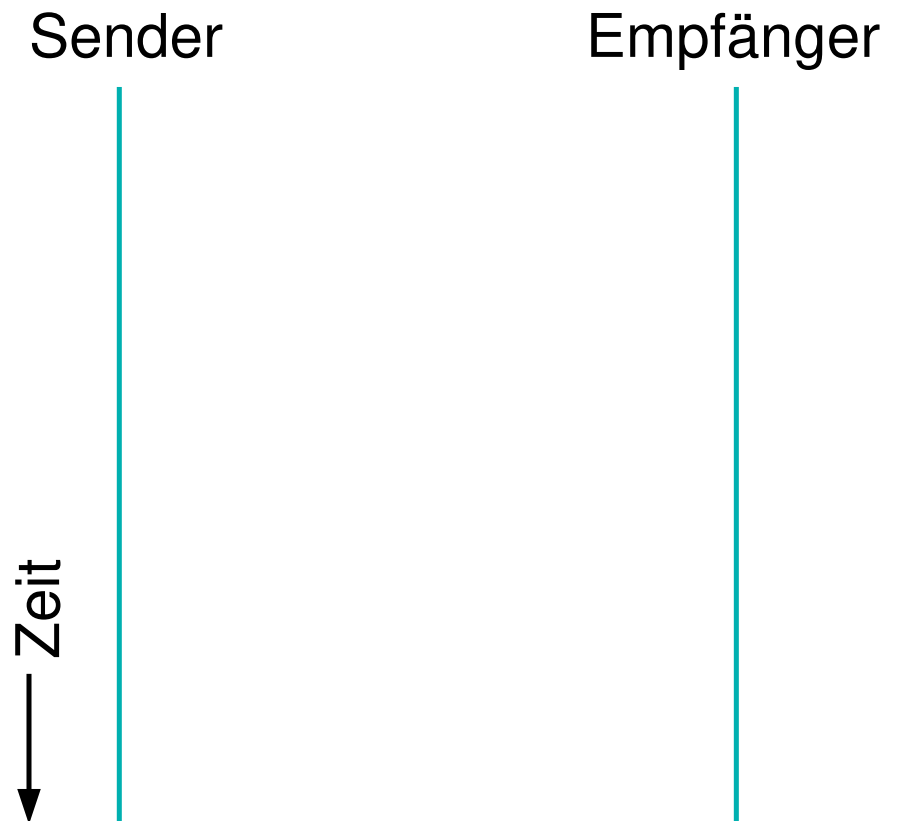
- ➔ Bei Stop-and-Wait reicht eine 1 Bit lange Sequenznummer
 - ➔ d.h. abwechselnd 0 und 1
 - ➔ Voraussetzung: Leitung vertauscht keine Frames



Motivation

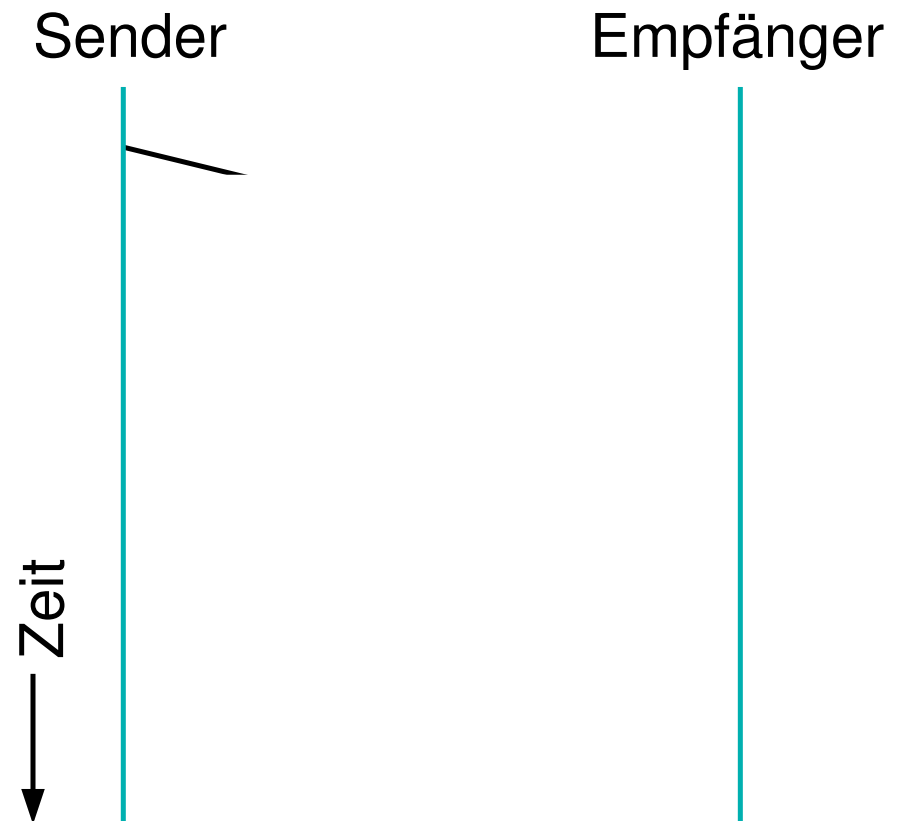
- ➔ Problem bei *Stop-and-Wait*:
 - ➔ Leitung wird nicht ausgelastet, da nur ein Frame pro RTT übertragen werden kann

- ➔ Um Leitung auszulasten:
 - ➔ Sender sollte die Datenmenge senden, die dem Verzögerungs(RTT)-Bandbreiten-Produkt entspricht, bevor er auf das erste ACK wartet



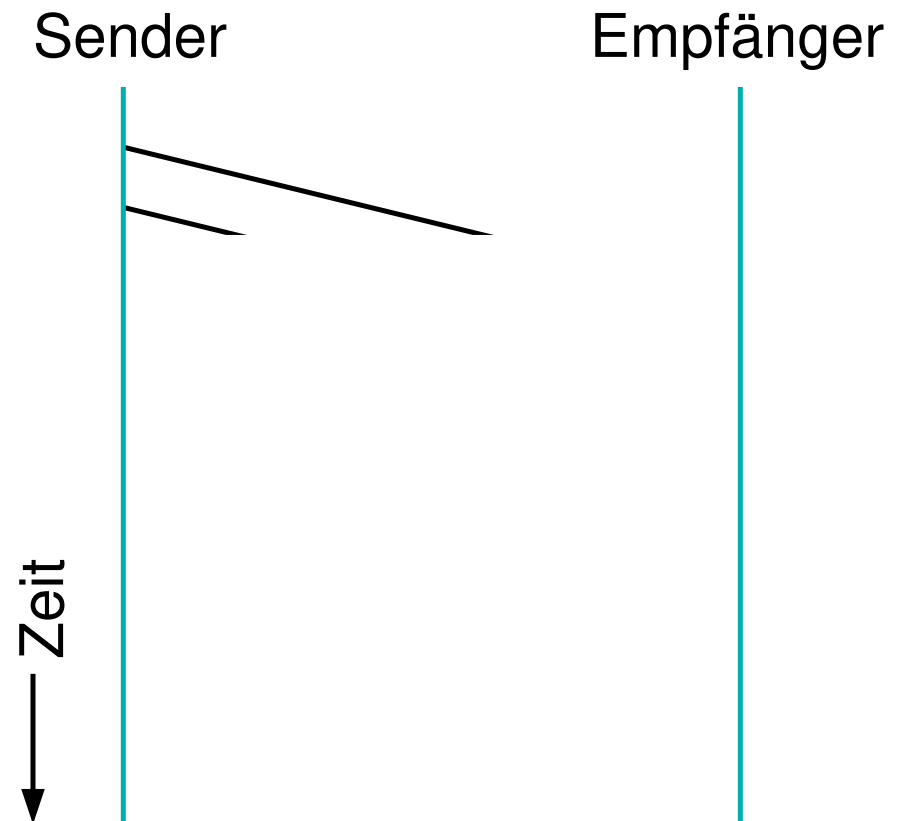
Motivation

- ➔ Problem bei *Stop-and-Wait*:
 - ➔ Leitung wird nicht ausgelastet, da nur ein Frame pro RTT übertragen werden kann
- ➔ Um Leitung auszulasten:
 - ➔ Sender sollte die Datenmenge senden, die dem Verzögerungs(RTT)-Bandbreiten-Produkt entspricht, bevor er auf das erste ACK wartet



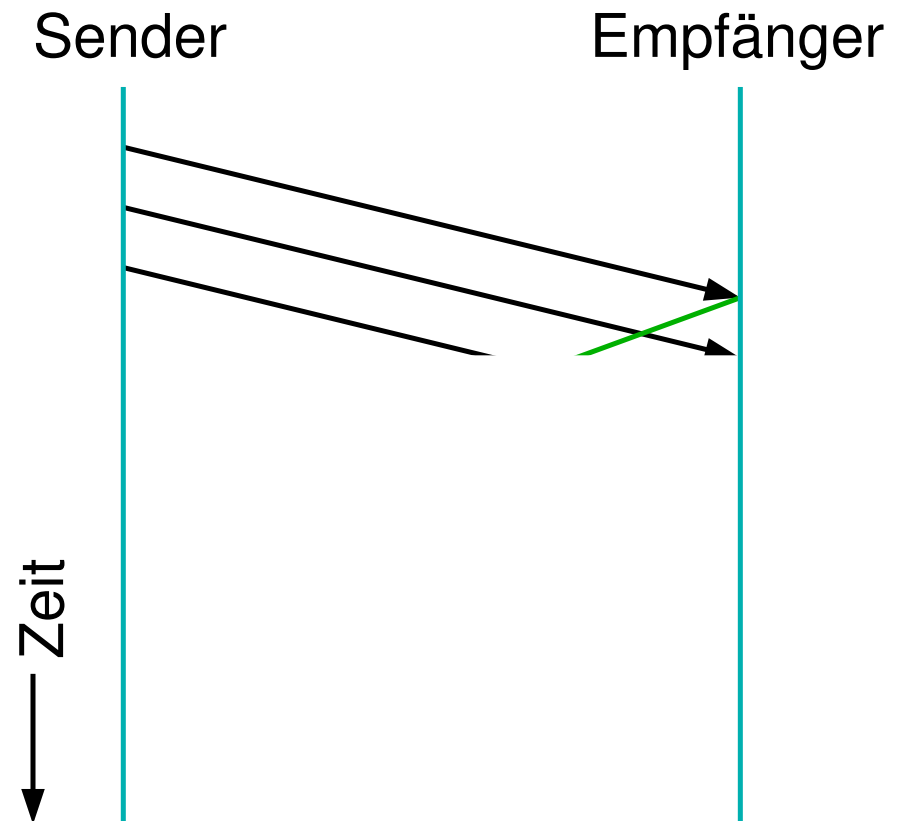
Motivation

- ➔ Problem bei *Stop-and-Wait*:
 - ➔ Leitung wird nicht ausgelastet, da nur ein Frame pro RTT übertragen werden kann
- ➔ Um Leitung auszulasten:
 - ➔ Sender sollte die Datenmenge senden, die dem Verzögerungs(RTT)-Bandbreiten-Produkt entspricht, bevor er auf das erste ACK wartet



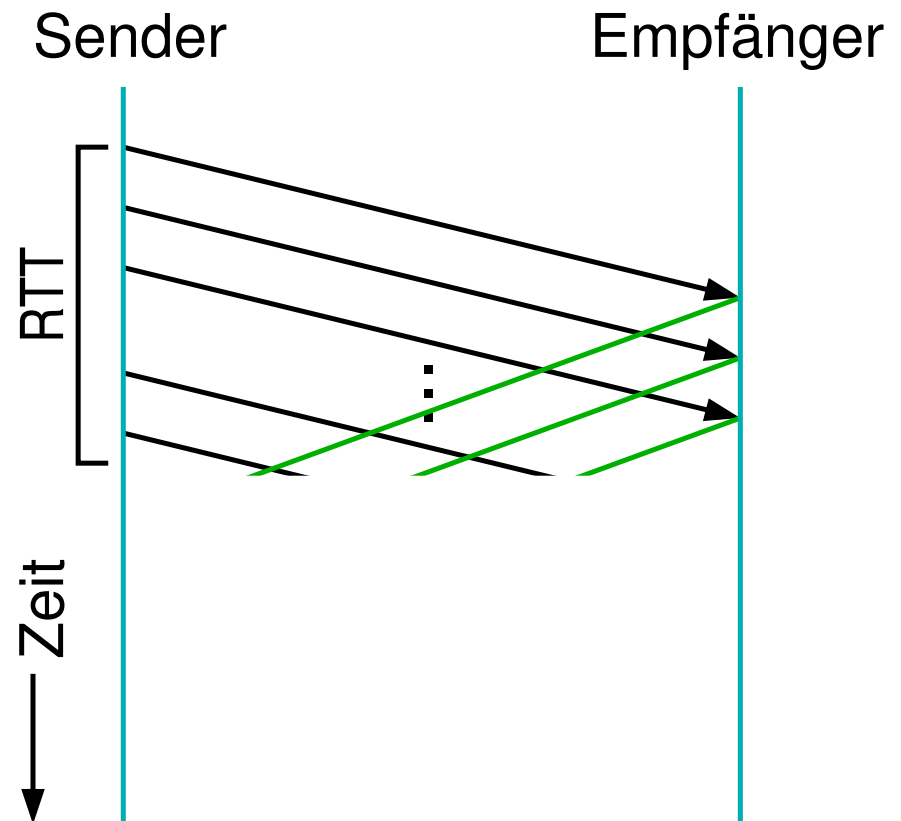
Motivation

- ➔ Problem bei *Stop-and-Wait*:
 - ➔ Leitung wird nicht ausgelastet, da nur ein Frame pro RTT übertragen werden kann
- ➔ Um Leitung auszulasten:
 - ➔ Sender sollte die Datenmenge senden, die dem Verzögerungs(RTT)-Bandbreiten-Produkt entspricht, bevor er auf das erste ACK wartet



Motivation

- ➔ Problem bei *Stop-and-Wait*:
 - ➔ Leitung wird nicht ausgelastet, da nur ein Frame pro RTT übertragen werden kann
- ➔ Um Leitung auszulasten:
 - ➔ Sender sollte die Datenmenge senden, die dem Verzögerungs(RTT)-Bandbreiten-Produkt entspricht, bevor er auf das erste ACK wartet



Motivation

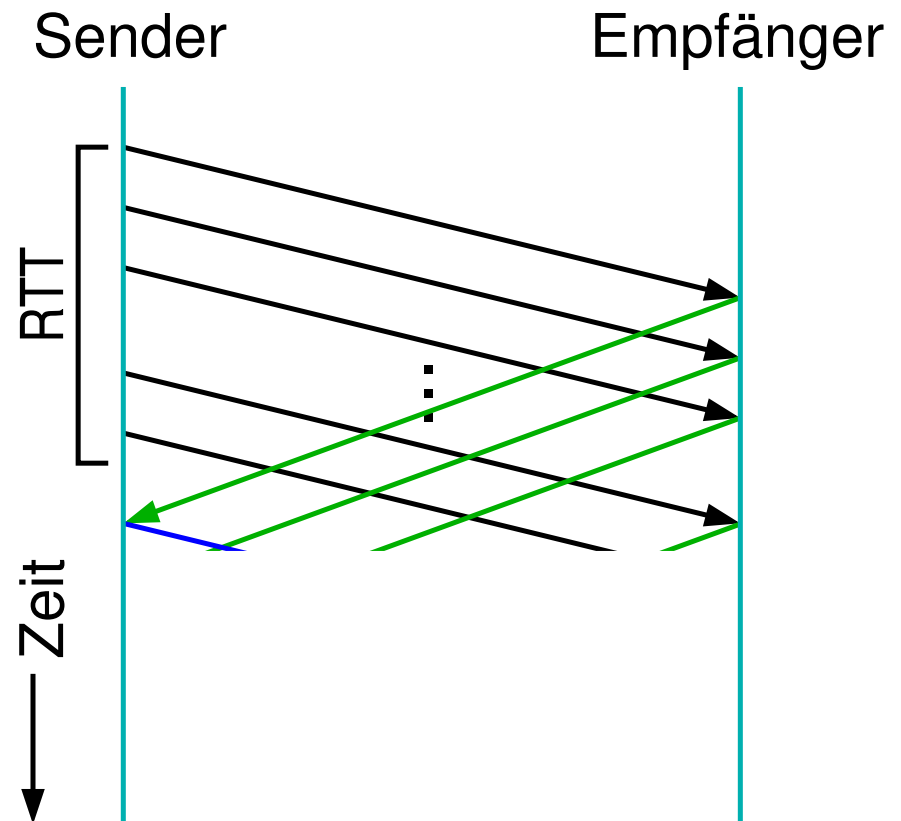
➔ Problem bei *Stop-and-Wait*:

➔ Leitung wird nicht ausgelastet, da nur ein Frame pro RTT übertragen werden kann

➔ Um Leitung auszulasten:

➔ Sender sollte die Datenmenge senden, die dem Verzögerungs(RTT)-Bandbreiten-Produkt entspricht, bevor er auf das erste ACK wartet

➔ dann mit jedem ACK einen neuen Frame senden



Motivation

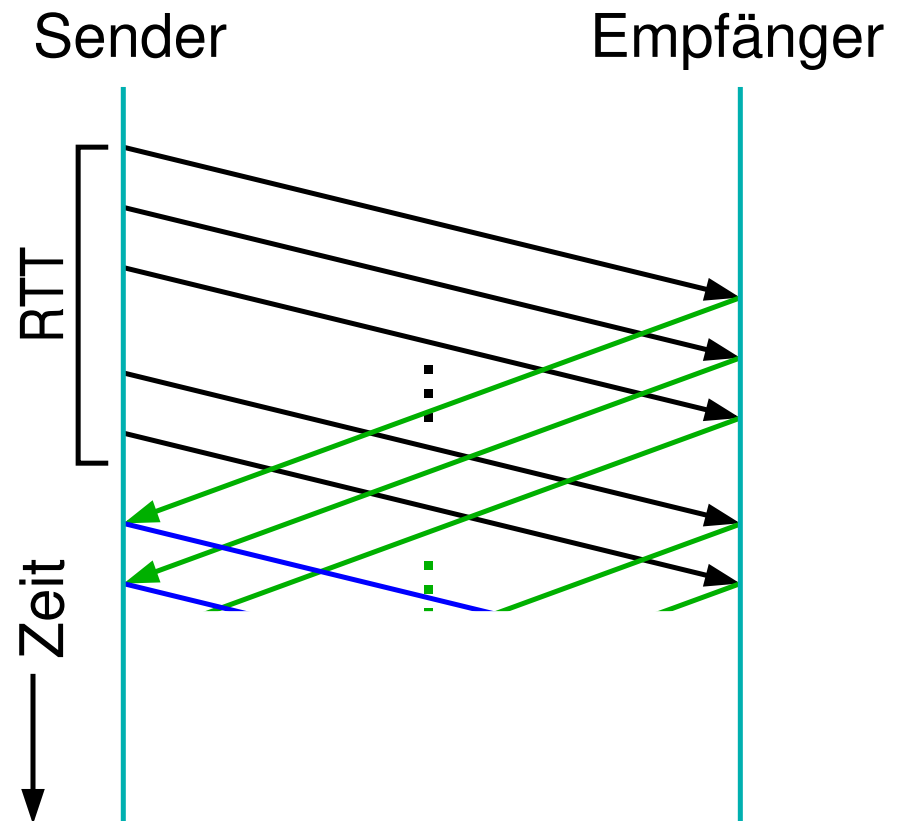
➔ Problem bei *Stop-and-Wait*:

➔ Leitung wird nicht ausgelastet, da nur ein Frame pro RTT übertragen werden kann

➔ Um Leitung auszulasten:

➔ Sender sollte die Datenmenge senden, die dem Verzögerungs(RTT)-Bandbreiten-Produkt entspricht, bevor er auf das erste ACK wartet

➔ dann mit jedem ACK einen neuen Frame senden



Motivation

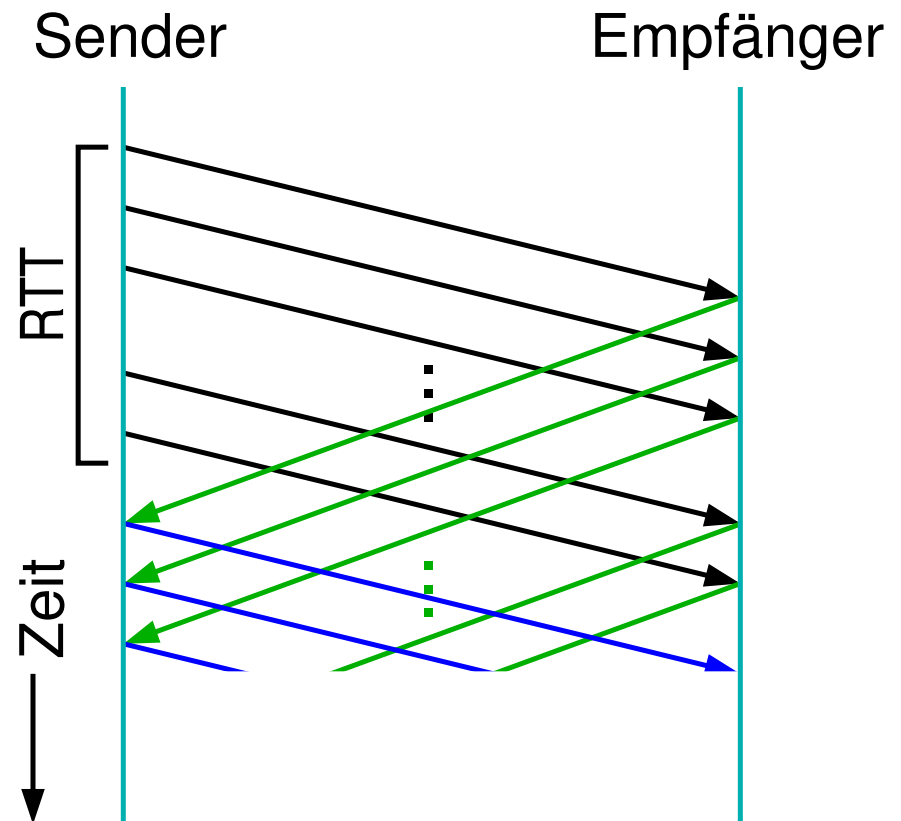
➔ Problem bei *Stop-and-Wait*:

➔ Leitung wird nicht ausgelastet, da nur ein Frame pro RTT übertragen werden kann

➔ Um Leitung auszulasten:

➔ Sender sollte die Datenmenge senden, die dem Verzögerungs(RTT)-Bandbreiten-Produkt entspricht, bevor er auf das erste ACK wartet

➔ dann mit jedem ACK einen neuen Frame senden



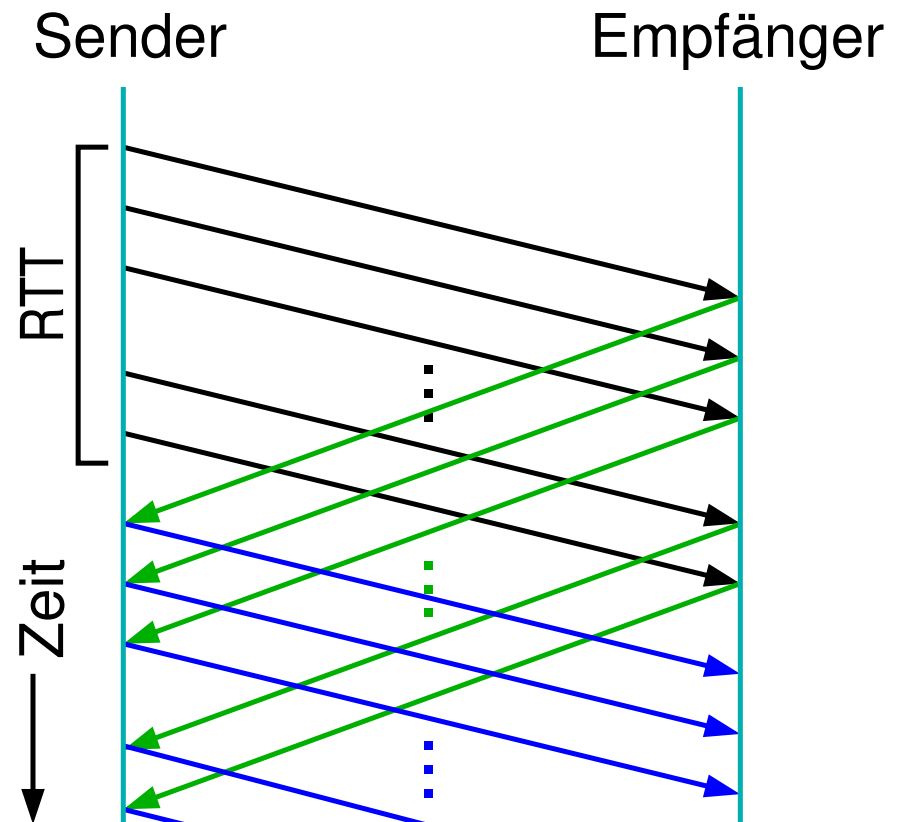
Motivation

➔ Problem bei *Stop-and-Wait*:

➔ Leitung wird nicht ausgelastet, da nur ein Frame pro RTT übertragen werden kann

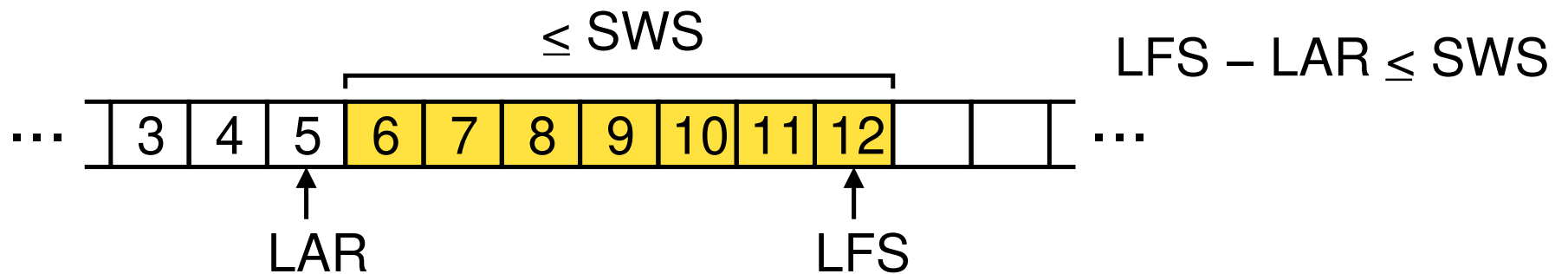
➔ Um Leitung auszulasten:

- ➔ Sender sollte die Datenmenge senden, die dem Verzögerungs(RTT)-Bandbreiten-Produkt entspricht, bevor er auf das erste ACK wartet
- ➔ dann mit jedem ACK einen neuen Frame senden



Funktionsweise

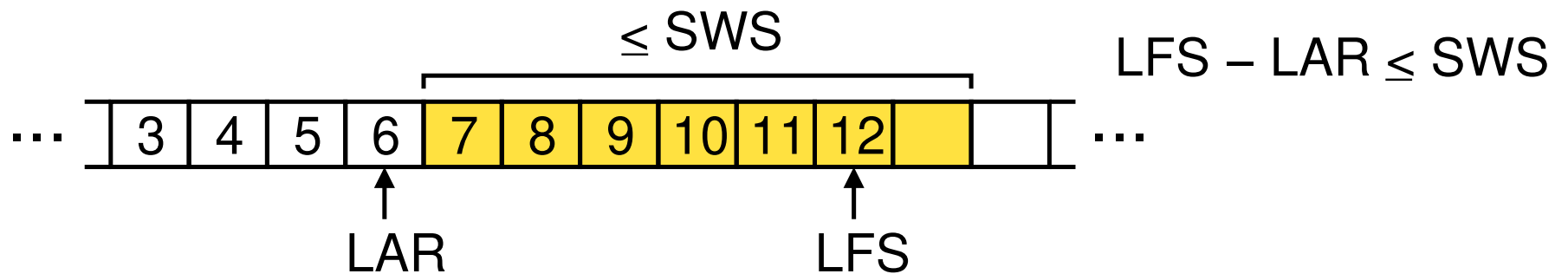
- ➔ Jeder Frame erhält eine Sequenznummer
- ➔ Der **Sender** besitzt ein „Schiebefenster“ (*Sliding Window*):



- ➔ Jeder Eintrag steht für einen gesendeten Frame
- ➔ LAR: *Last Acknowledgement Received*
 - ➔ bis zu diesem Frame (incl.) wurden alle quittiert
- ➔ LFS: *Last Frame Sent*
- ➔ SWS: *Sender Window Size*
 - ➔ max. SWS Frames werden ohne ACK abgeschickt

Funktionsweise

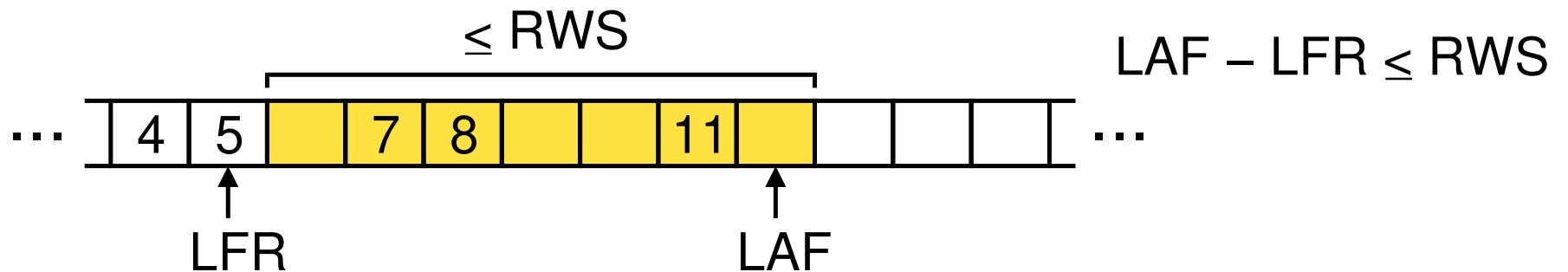
- ➔ Jeder Frame erhält eine Sequenznummer
- ➔ Der **Sender** besitzt ein „Schiebefenster“ (*Sliding Window*):



- ➔ Jeder Eintrag steht für einen gesendeten Frame
- ➔ LAR: *Last Acknowledgement Received*
 - ➔ bis zu diesem Frame (incl.) wurden alle quittiert
- ➔ LFS: *Last Frame Sent*
- ➔ SWS: *Sender Window Size*
 - ➔ max. SWS Frames werden ohne ACK abgeschickt

Funktionsweise ...

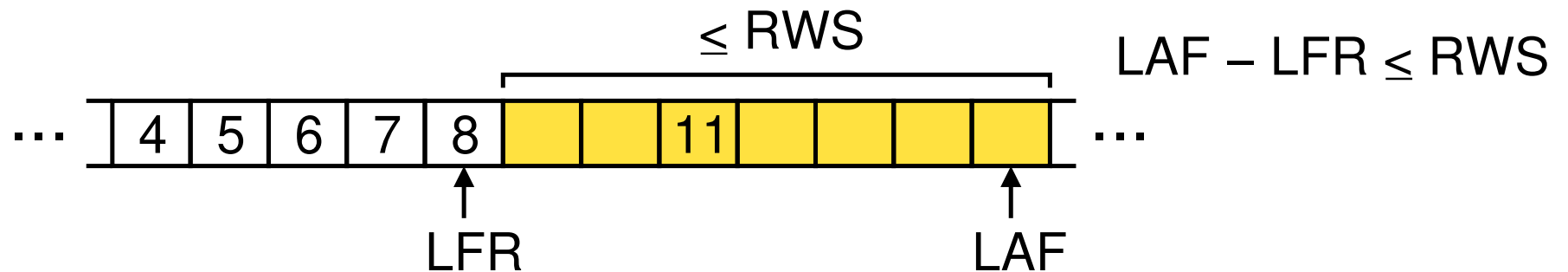
➔ Der **Empfänger** hat ebenfalls ein *Sliding Window*:



- ➔ Jeder Eintrag steht für einen empfangenen Frame
- ➔ LFR: *Last Frame Received*
 - ➔ alle Frames n mit $n \leq LFR$ wurden korrekt empfangen und quittiert
- ➔ LAF: *Largest Acceptable Frame*
 - ➔ Frame n wird nur akzeptiert, wenn $LFR < n \leq LAF$
- ➔ RWS: *Receiver Window Size*
 - ➔ Anzahl der Pufferplätze beim Empfänger

Funktionsweise ...

➔ Der **Empfänger** hat ebenfalls ein *Sliding Window*:



- ➔ Jeder Eintrag steht für einen empfangenen Frame
- ➔ LFR: *Last Frame Received*
 - ➔ alle Frames n mit $n \leq LFR$ wurden korrekt empfangen und quittiert
- ➔ LAF: *Largest Acceptable Frame*
 - ➔ Frame n wird nur akzeptiert, wenn $LFR < n \leq LAF$
- ➔ RWS: *Receiver Window Size*
 - ➔ Anzahl der Pufferplätze beim Empfänger

Quittierung von Frames

- ➔ **Akkumulatives Acknowledgement:**
 - ➔ ACK für Frame n gilt auch für alle Frames $\leq n$
- ➔ Zusätzlich **negative Acknowledgements** möglich:
 - ➔ Wenn Frame n empfangen wird,
aber Frame m mit $m < n$ noch aussteht,
wird für Frame m ein NACK geschickt
- ➔ Alternative: **selektives Acknowledgement:**
 - ➔ ACK für Frame n gilt nur für diesen Frame

Problem in der Praxis

- ➔ Begrenzte Anzahl von Bits für die Sequenznummer im Frame-Header
 - ➔ z.B. bei 3 Bits nur Nummern 0 ... 7 möglich
- ➔ Reicht ein endlicher Bereich an Sequenznummern aus?
 - ➔ ja, abhängig von SWS und RWS:
 - ➔ falls $RWS = 1$: $NSeqNum \geq SWS + 1$
 - ➔ falls $RWS = SWS$: $NSeqNum \geq 2 \cdot SWS$
 - ($NSeqNum$ = Anzahl von Sequenznummern)
 - ➔ aber nur, wenn die Reihenfolge der Frames bei der Übertragung nicht verändert werden kann!



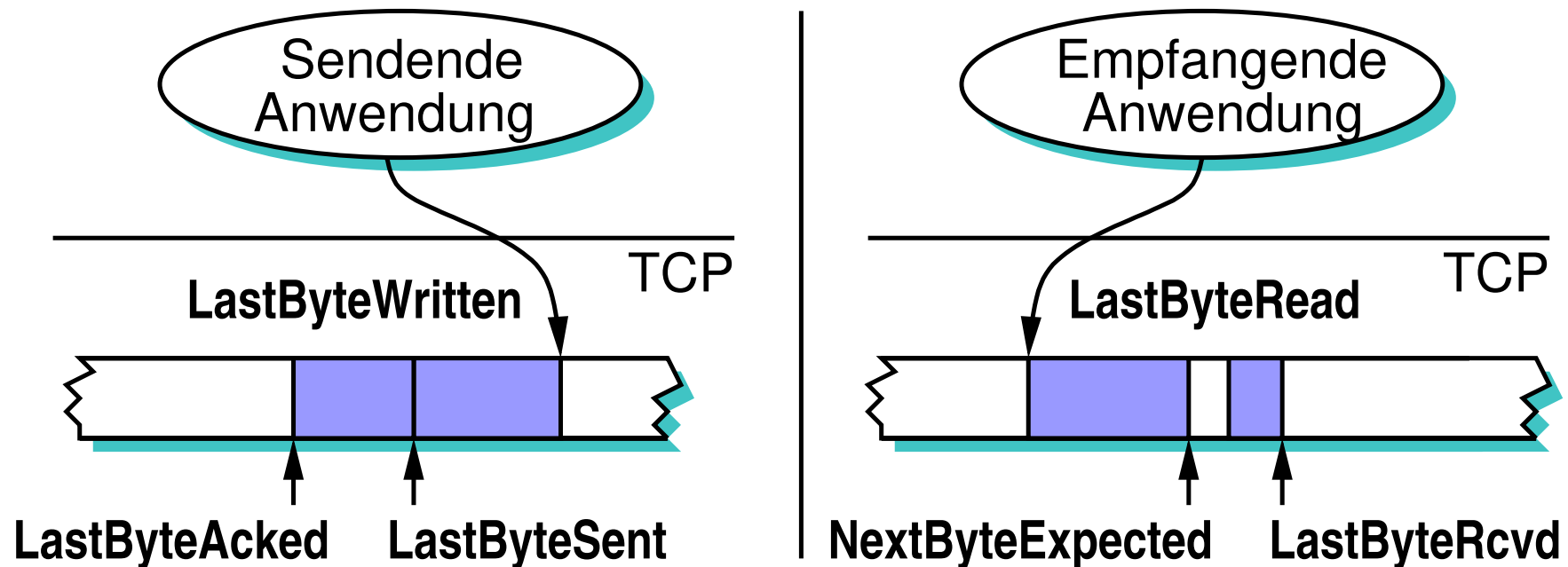
- ➔ TCP nutzt den *Sliding-Window*-Algorithmus
- ➔ Prinzipiell wie in 7.4.2 vorgestellt, aber Unterschiede:
 - ➔ Sequenznummer zählt Bytes, nicht Segmente
 - ➔ TCP benötigt Verbindungsaufbau und -abbau
 - ➔ Austausch der *Sliding-Window* Parameter
 - ➔ Netzwerk (IP) kann Pakete umordnen
 - ➔ TCP toleriert bis zu 120 Sekunden alte Pakete
 - ➔ Keine feste Fenstergröße
 - ➔ Sendefenstergröße angepasst an Puffer des Empfängers bzw. Lastsituation im Netz
 - ➔ RTT ist nicht konstant, sondern ändert sich laufend
 - ➔ Timeout muß adaptiv sein

Aufgaben des Sliding-Window-Algorithmus in TCP

- ➔ Zuverlässige Übertragung
- ➔ Sicherstellung der richtigen Reihenfolge der Segmente
 - ➔ TCP gibt Segmente nur dann an obere Schicht weiter, wenn alle vorherigen Segmente bestätigt wurden
- ➔ Flußkontrolle
 - ➔ keine feste Sendefenstergröße
 - ➔ Empfänger teilt dem Sender den freien Pufferplatz mit (**AdvertisedWindow**)
 - ➔ Sender passt Sendefenstergröße entsprechend an
- ➔ Überlastkontrolle
 - ➔ Sendefenstergröße wird dynamisch an Netzlast angepasst

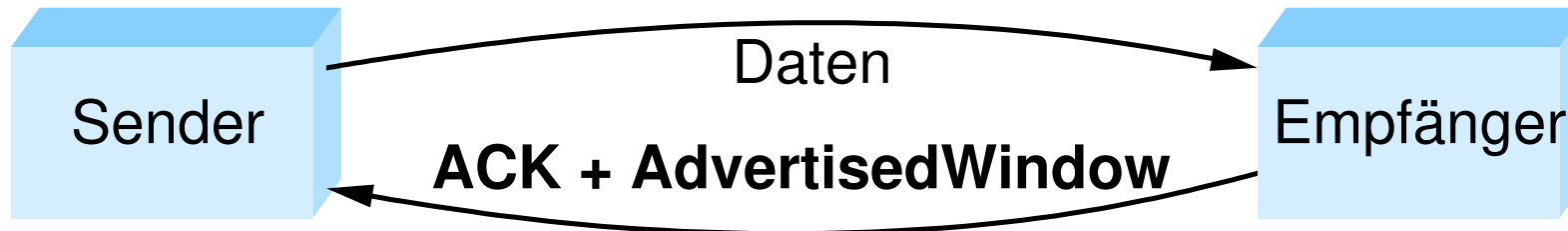
Zuverlässige und geordnete Übertragung

- ➔ Algorithmus arbeitet auf Byte-Ebene
- ➔ Sequenznummern werden um die Anzahl gesendeter bzw. empfangener Bytes erhöht



Flußkontrolle

- ➔ Empfänger teilt Sender die Größe des freien Puffers mit:



- ➔ **AdvertisedWindow =**
MaxRcvBuffer – (LastByteRcvd – LastByteRead)
- ➔ Sender muß sicherstellen, daß jederzeit gilt:
 - ➔ **LastByteSent – LastByteAcked ≤ AdvertisedWindow**
- ➔ Differenz: Datenmenge, die der Sender noch senden kann
- ➔ Sendende Anwendung wird blockiert, wenn Daten (**y** Bytes) nicht mehr in Sendepuffer passen, d.h. wenn
 - ➔ **LastByteWritten – LastByteAcked + y > MaxSendBuffer**

Sequenznummern-Überlauf

- ➔ Erinnerung an **7.4.2**: endlicher Sequenznummernbereich nur möglich, wenn Netzwerk die Reihenfolge erhält
- ➔ TCP-Header: 32-Bit Feld für Sequenznummern
- ➔ Pakete können bis zu 120 Sekunden alt werden

Bandbreite	Zeit bis zum Überlauf
10 MBit/s (Ethernet)	57 Minuten
100 MBit/s (FDDI)	6 Minuten
155 MBit/s (OC-3)	4 Minuten
1,2 GBit/s (OC-24)	28 Sekunden
9,95 GBit/s (OC-192)	3,4 Sekunden

- ➔ ⇒ TCP-Erweiterung: Zeitstempel als Überlaufschutz

Größe des AdvertisedWindow

- ➔ TCP-Header sieht 16-Bit vor, d.h. max. 64 KBytes
- ➔ Nötige Sendefenster-Größe, um Kanal gefüllt zu halten, bei RTT = 100 ms (z.B. Transatlantik-Verbindung):

Bandbreite	RTT * Bandbreite
10 MBit/s (Ethernet)	122 KByte
100 MBit/s (FDDI)	1,2 MByte
155 MBit/s (OC-3)	1,8 MByte
1,2 GBit/s (OC-24)	14,8 MByte
9,95 GBit/s (OC-192)	119 MByte

- ➔ ⇒ TCP-Erweiterung: Festlegung eines Skalierungsfaktors für **AdvertisedWindow** beim Verbindungsaufbau

Adaptive Neuübertragung

- ➔ Timeout für Neuübertragung muß abhängig von RTT gewählt werden
- ➔ Im Internet: RTT ist unterschiedlich und veränderlich
- ➔ Daher: adaptive Bestimmung des Timeouts nötig
 - ➔ ursprünglich:
 - ➔ Messung der durchschnittlichen RTT (Zeit zwischen Senden eines Segments und Ankunft des ACK)
 - ➔ $\text{Timeout} = 2 \cdot \text{durchschnittliche RTT}$
 - ➔ Problem:
 - ➔ Varianz der RTT-Meßwerte nicht berücksichtigt
 - ➔ bei hoher Varianz sollte der Timeout deutlich über dem Mittelwert liegen

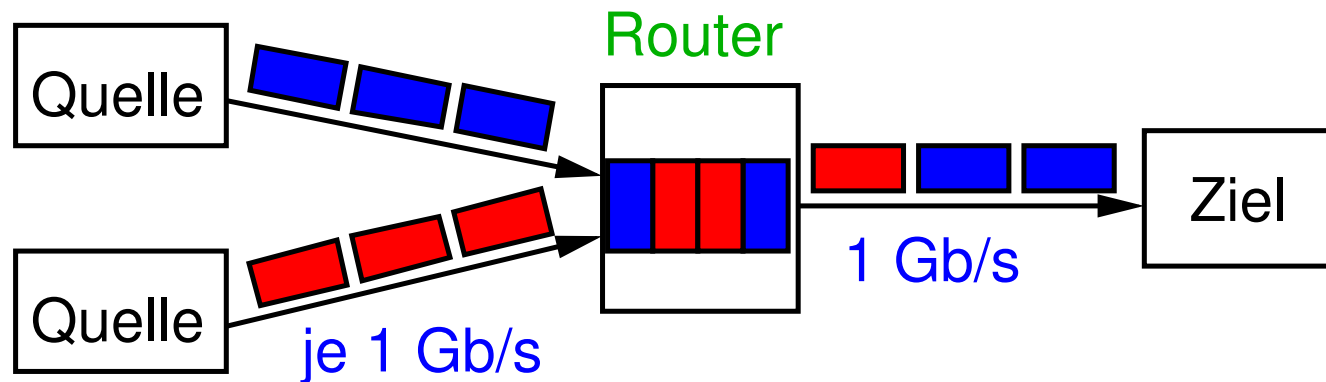
Adaptive Neuübertragung: Jacobson/Karels-Algorithmus

- ➔ Berechne gleitenden Mittelwert und (approximierte) Standardabweichung der RTT:
 - ➔ **Deviation** = $\delta \cdot |\text{SampleRTT} - \text{EstimatedRTT}| + (1 - \delta) \cdot \text{Deviation}$
 - ➔ **EstimatedRTT** = $\delta \cdot \text{SampleRTT} + (1 - \delta) \cdot \text{EstimatedRTT}$
- ➔ Berücksichtige Standardabweichung bei Timeout-Berechnung:
 - ➔ **TimeOut** = $\mu \cdot \text{EstimatedRTT} + \Phi \cdot \text{Deviation}$
- ➔ Typisch: $\mu = 1$, $\Phi = 4$, $\delta = 0,125$

Was bedeutet Überlast?

- ➔ Pakete konkurrieren um Bandbreite einer Verbindung
- ➔ Bei unzureichender Bandbreite:
 - ➔ Puffern der Pakete im Router
- ➔ Bei Pufferüberlauf:
 - ➔ Pakete verwerfen
- ➔ Ein Netzwerk mit häufigem Pufferüberlauf heißt **überlastet** (*congested*)

Beispiel einer Überlastsituation



- ➔ Sender können das Problem nicht direkt erkennen
- ➔ Adaptives Routing löst das Problem nicht, trotz schlechter Link-Metrik für überlastete Leitung
 - ➔ verschiebt Problem nur an andere Stelle
 - ➔ Umleitung ist nicht immer möglich (evtl. nur ein Weg)
 - ➔ im Internet wegen Komplexität derzeit utopisch

Überlastkontrolle

- ➔ Erkennen und möglichst schnelles Beenden der Überlast
 - ➔ z.B. einige Sender mit hoher Datenrate stoppen
 - ➔ in der Regel aber Fairness gewünscht

Überlastvermeidung

- ➔ Erkennen von drohenden Überlastsituationen und Vermeidung der Überlast (☞ **Rechnernetze II**)

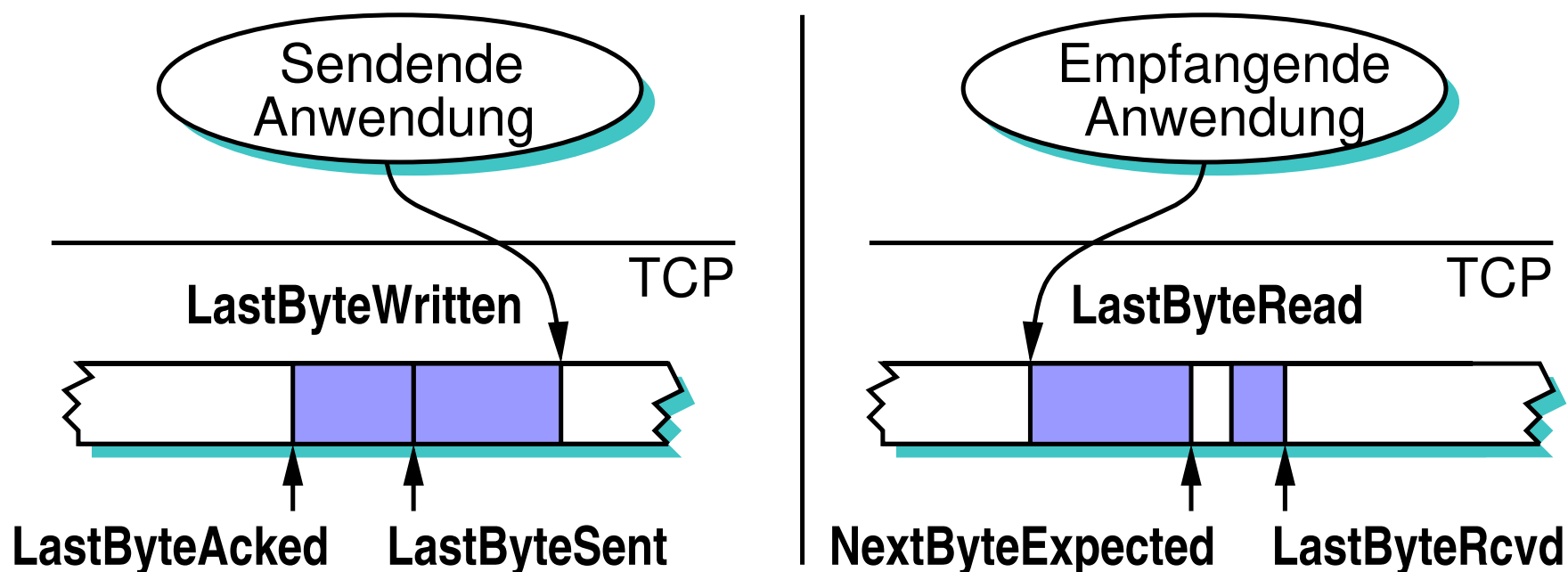
Abgrenzung

- ➔ **Flußkontrolle** verhindert, daß **ein Sender** seinen **Empfänger** überlastet
- ➔ **Überlastkontrolle** (bzw. **-vermeidung**) verhindert, daß **mehrere Sender** einen Teil des **Netzwerks** (= Zwischenknoten) überlasten

- ➔ Einführung Ende der 1980'er Jahre (8 Jahre nach Einführung von TCP) zur Behebung akuter Überlastprobleme
 - ➔ Überlast \Rightarrow Paketverlust \Rightarrow Neuübertragung \Rightarrow noch mehr Überlast!
- ➔ Idee:
 - ➔ jeder Sender bestimmt, für wieviele Pakete (Segmente) Platz im Netzwerk ist
 - ➔ wenn Netz „gefüllt“ ist:
 - ➔ Ankunft eines ACKs \Rightarrow Senden eines neuen Pakets
 - ➔ „selbsttaktend“
- ➔ Problem: Bestimmung der (momentanen) Kapazität
 - ➔ dauernder Auf- und Abbau anderer Verbindungen

Erinnerung: Flußkontrolle mit *Sliding-Window-Algorithmus*

- ➔ Empfänger sendet in ACKs **AdvertisedWindow** = **MaxRcvBuffer** – (**LastByteRcvd** – **LastByteRead**)
- ➔ Sender darf dann noch maximal so viele Bytes senden:
EffectiveWindow = **AdvertisedWindow** – (**LastByteSent** – **LastByteAcked**)





Erweiterung des *Sliding-Window*-Algorithmus

- ➔ Einführung eines **CongestionWindow**
 - ➔ Sender kann noch so viele Bytes senden, ohne Netzwerk zu überlasten
- ➔ Neue Berechnung für **EffectiveWindow**
 - ➔ $\text{MaxWindow} = \text{MIN}(\text{CongestionWindow}, \text{AdvertisedWindow})$
 - ➔ $\text{EffectiveWindow} = \text{MaxWindow} - (\text{LastByteSent} - \text{LastByteAcked})$
- ➔ Damit: weder Empfänger noch Netzwerk überlastet
- ➔ Frage: Bestimmung des **CongestionWindow**?



Bestimmung des CongestionWindow

- ➔ Hosts bestimmen **CongestionWindow** durch Beobachtung des Paketverlusts
- ➔ Basismechanismus:
 - ➔ *Additive Increase / Multiplicative Decrease*
- ➔ Erweiterungen:
 - ➔ *Slow Start*
 - ➔ *Fast Retransmit / Fast Recovery*

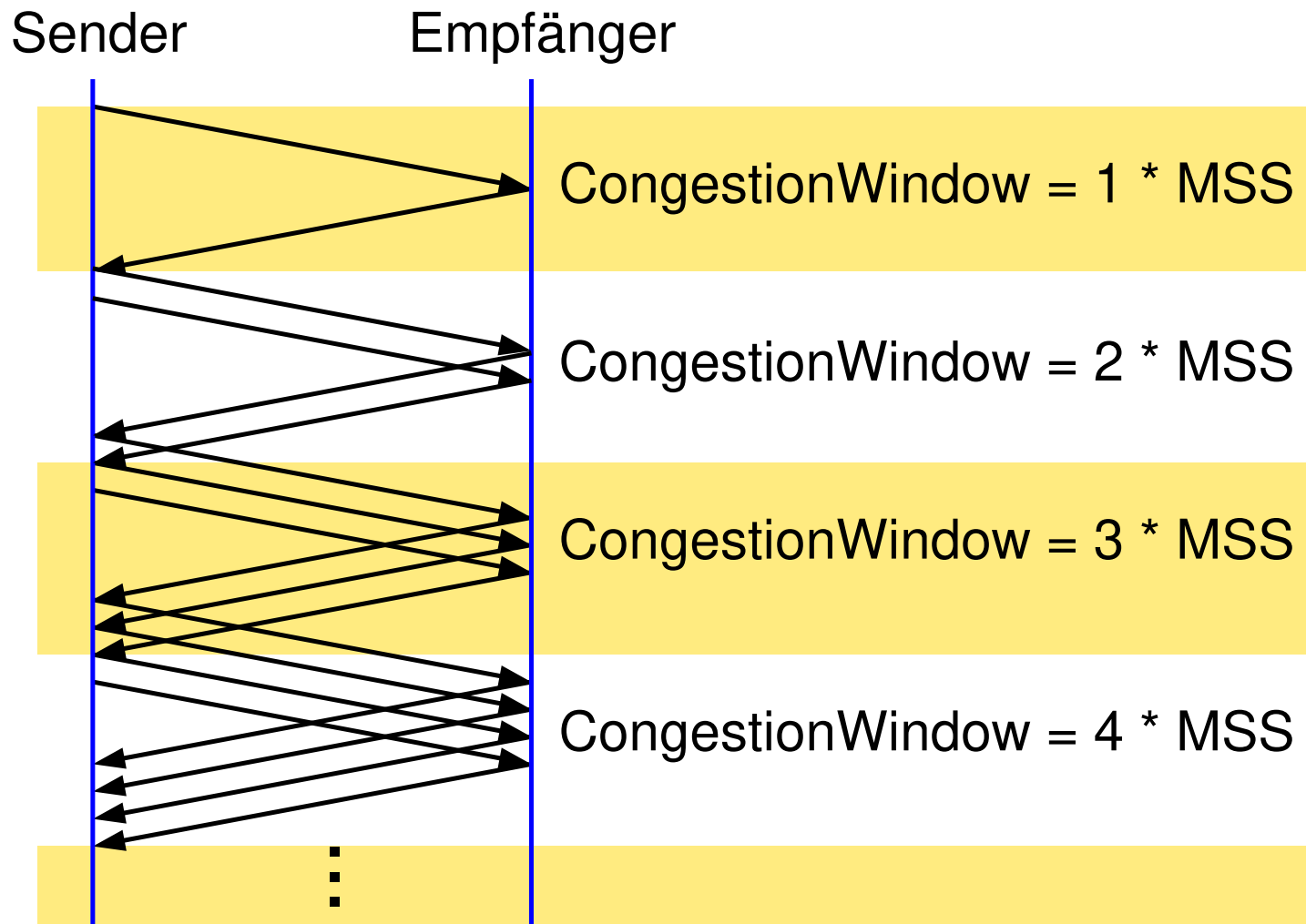


Vorgehensweise

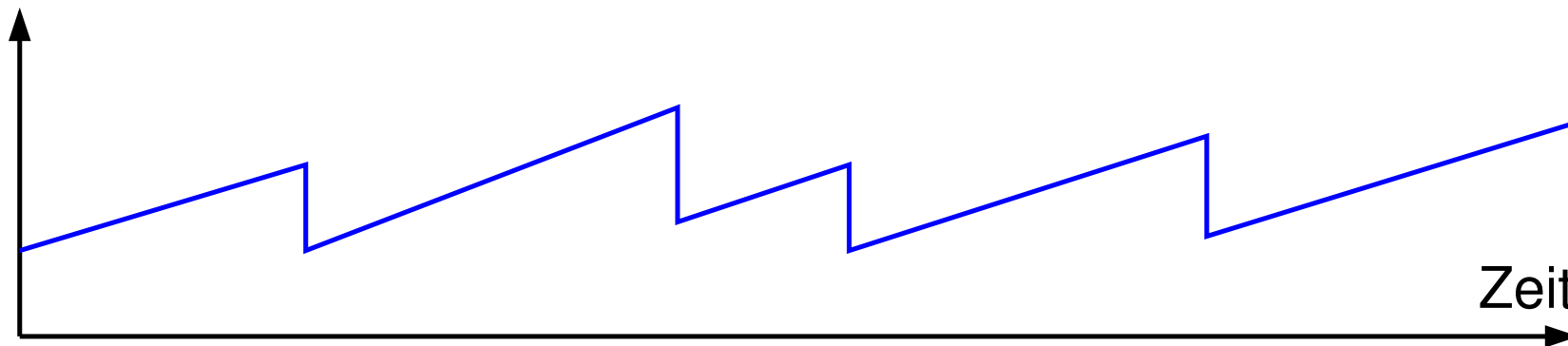
- ➔ **CongestionWindow** sollte
 - ➔ groß sein ohne / bei wenig Überlast
 - ➔ klein sein bei viel Überlast
- ➔ Überlast wird erkannt durch Paketverlust
- ➔ Bei Empfang eines ACK:
 - ➔ **Increment = MSS · (MSS / CongestionWindow)**
 - ➔ **CongestionWindow += Increment**

im Mittel: Erhöhung um MSS Bytes pro RTT
(MSS = *Maximum Segment Size* von TCP)
- ➔ Bei Timeout: **CongestionWindow** halbieren
 - ➔ höchstens, bis MSS erreicht ist

Additive Increase



Typischer Zeitverlauf des CongestionWindow



- ➡ Vorsichtige Erhöhung bei erfolgreicher Übertragung, drastische Reduzierung bei Erkennung einer Überlast
 - ➡ ausschlaggebend für Stabilität bei hoher Überlast
- ➡ Wichtig: gut angepaßte Timeout-Werte
 - ➡ Jacobson/Karels Algorithmus



Hintergrund

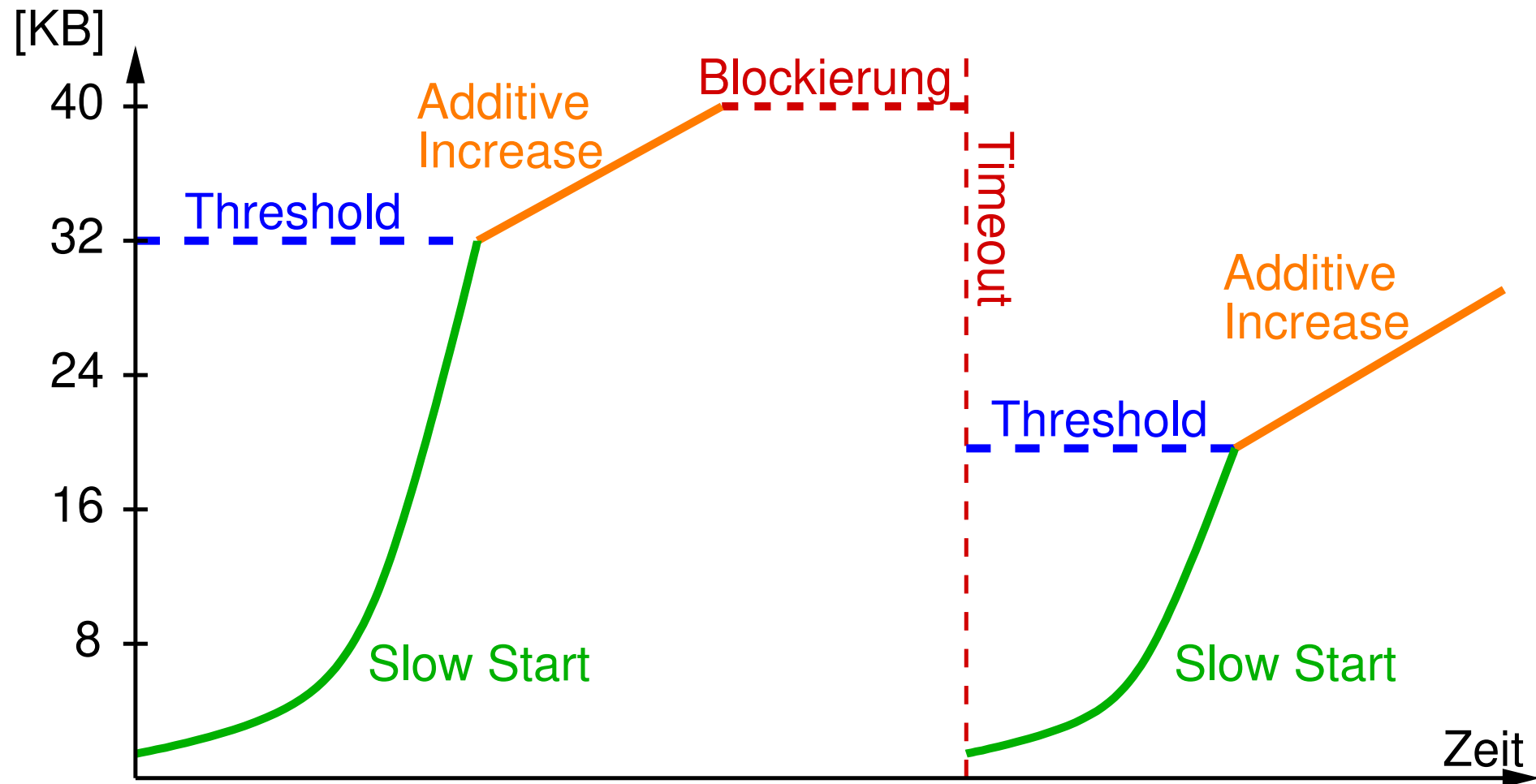
- ➔ Verhalten des ursprünglichen TCP beim Start (bzw. bei Wiederanlauf nach Timeout):
 - ➔ sende **AdvertisedWindow** an Daten (ohne auf ACKs zu warten)
 - ➔ d.h. Start mit maximalem **CongestionWindow**
- ➔ Zu aggressiv, kann zu hoher Überlast führen
- ➔ Andererseits: Start mit **CongestionWindow** = MSS und *Additive Increase* dauert zu lange
- ➔ Daher Mittelweg:
 - ➔ Start mit **CongestionWindow** = MSS
 - ➔ Verdopplung bis zum ersten Timeout



Verhalten bei Timeout

- ➔ *Slow Start* wird auch verwendet, wenn eine Verbindung bis zu einem Timeout blockiert:
 - ➔ Paket X geht verloren
 - ➔ Sendefenster ist ausgeschöpft, keine weiteren Pakete
 - ➔ nach Timeout: X wird neu übertragen, ein kumulatives ACK öffnet Sendefenster wieder
- ➔ In diesem Fall beim Timeout:
 - ➔ **CongestionThreshold** = **CongestionWindow** / 2
 - ➔ *Slow Start*, bis **CongestionThreshold** erreicht ist, danach *Additive Increase*

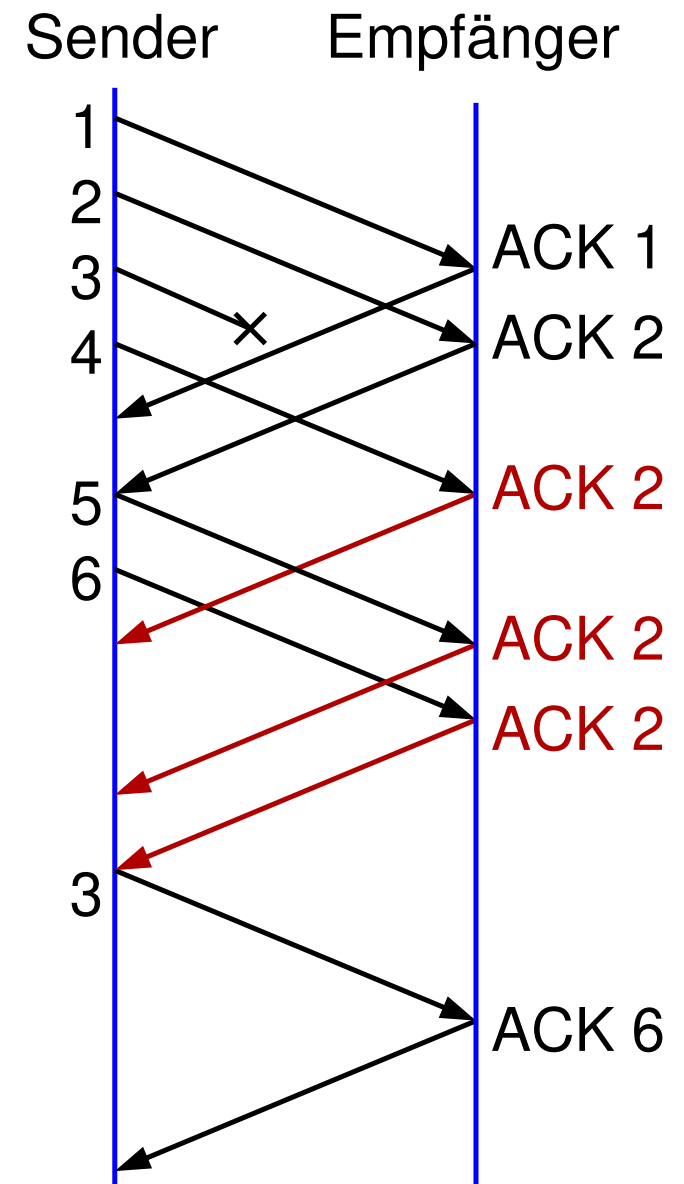
Typischer Verlauf des CongestionWindow



7.6.4 Fast Retransmit / Fast Recovery



- ➔ Lange Timeouts führen oft zum Blockieren der Verbindung
- ➔ Idee: Paketverlust kann auch durch Duplikat-ACKs erkannt werden
- ➔ Nach dem dritten Duplikat-ACK:
 - ➔ Paket erneut übertragen
 - ➔ **CongestionWindow** halbieren ohne *Slow Start*
- ➔ *Slow Start* nur noch am Anfang und bei wirklichem Timeout



Verbindungsaufbau

- ➔ Asymmetrisch:
 - ➔ Client (rufender Teilnehmer): **aktives Öffnen**
 - ➔ sende Verbindungswunsch zum Server
 - ➔ Server (gerufener Teilnehmer): **passives Öffnen**
 - ➔ warte auf eingehende Verbindungswünsche
 - ➔ akzeptiere ggf. einen Verbindungswunsch

Verbindungsabbau

- ➔ Symmetrisch:
 - ➔ beide Seiten müssen die Verbindung schließen



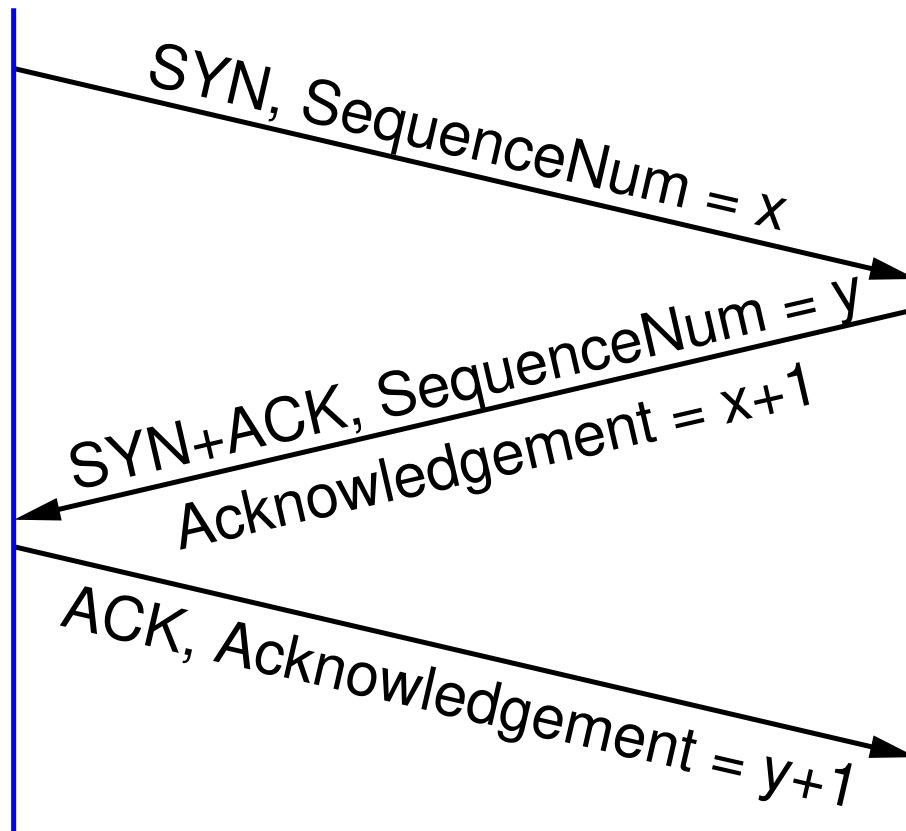
Zum Begriff der TCP-Verbindung

- ➔ Das Tupel (Quell-IP-Adresse, Quell-Port, Ziel-IP-Adresse, Ziel-Port) kennzeichnet eine TCP-Verbindung eindeutig
 - ➔ Nutzung als Demultiplex-Schlüssel
- ➔ Nach Abbau einer Verbindung und Wiederaufbau mit denselben IP-Adressen und Port-Nummern:
 - ➔ neue **Inkarnation** derselben Verbindung

Verbindungsaufbau: *Three-Way Handshake*

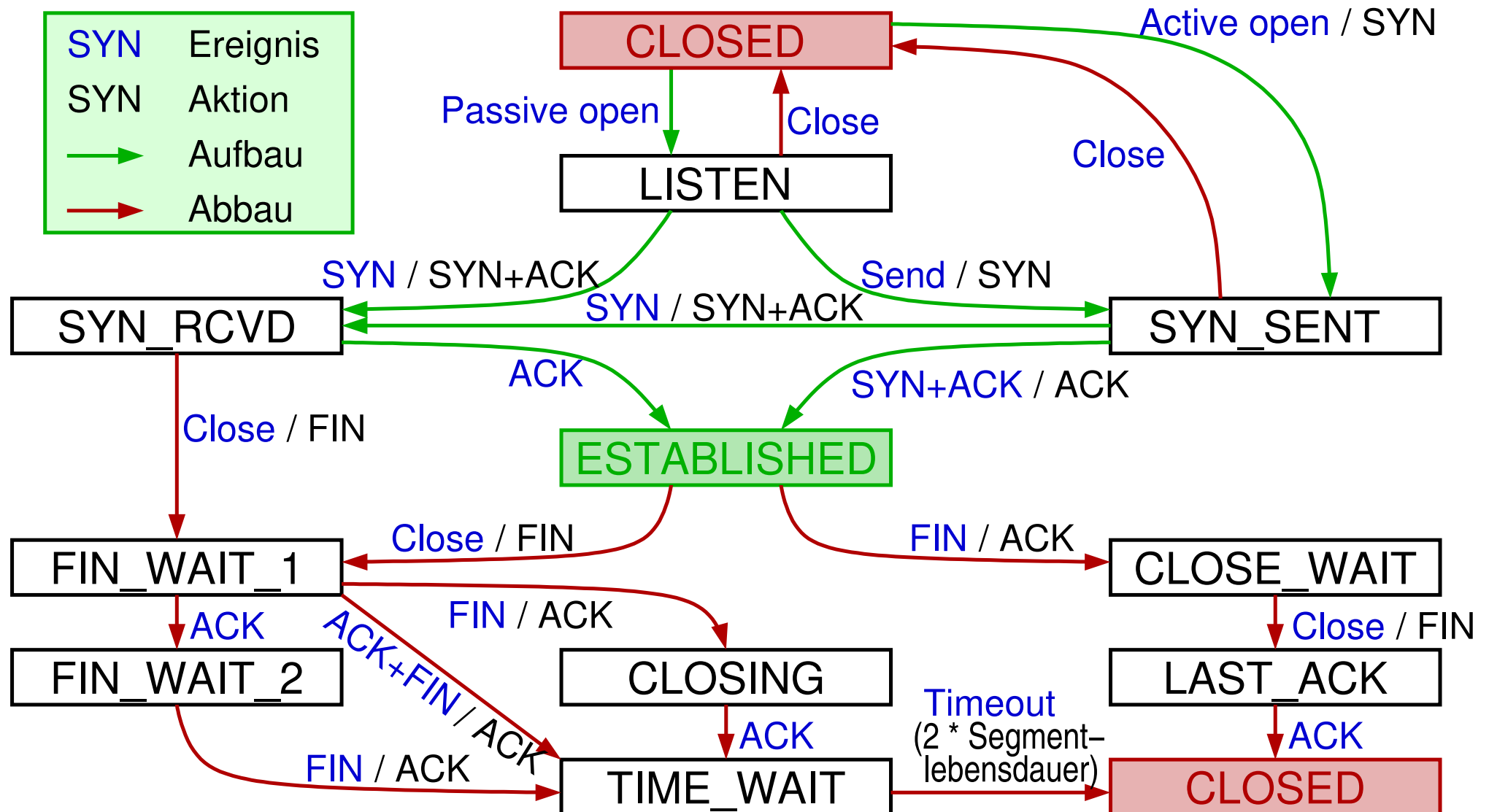
Aktiver
Teilnehmer
(Client)

Passiver
Teilnehmer
(Server)

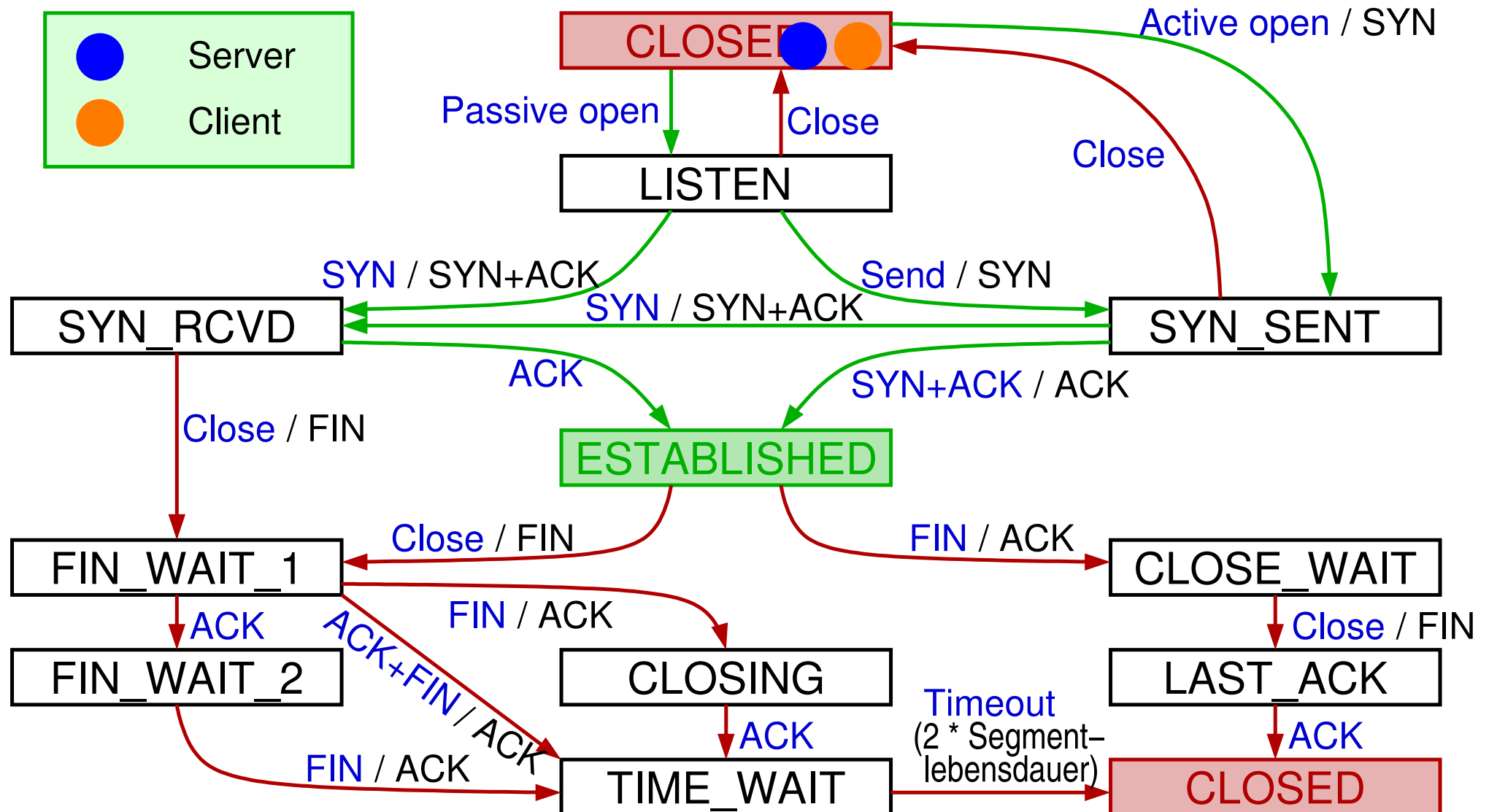


- ➔ Austausch von Sequenznummern
- ➔ „Zufälliger“ Startwert
 - ➔ jede Inkarnation nimmt andere Nummern
- ➔ Acknowledgement: nächste erwartete Sequenznummer

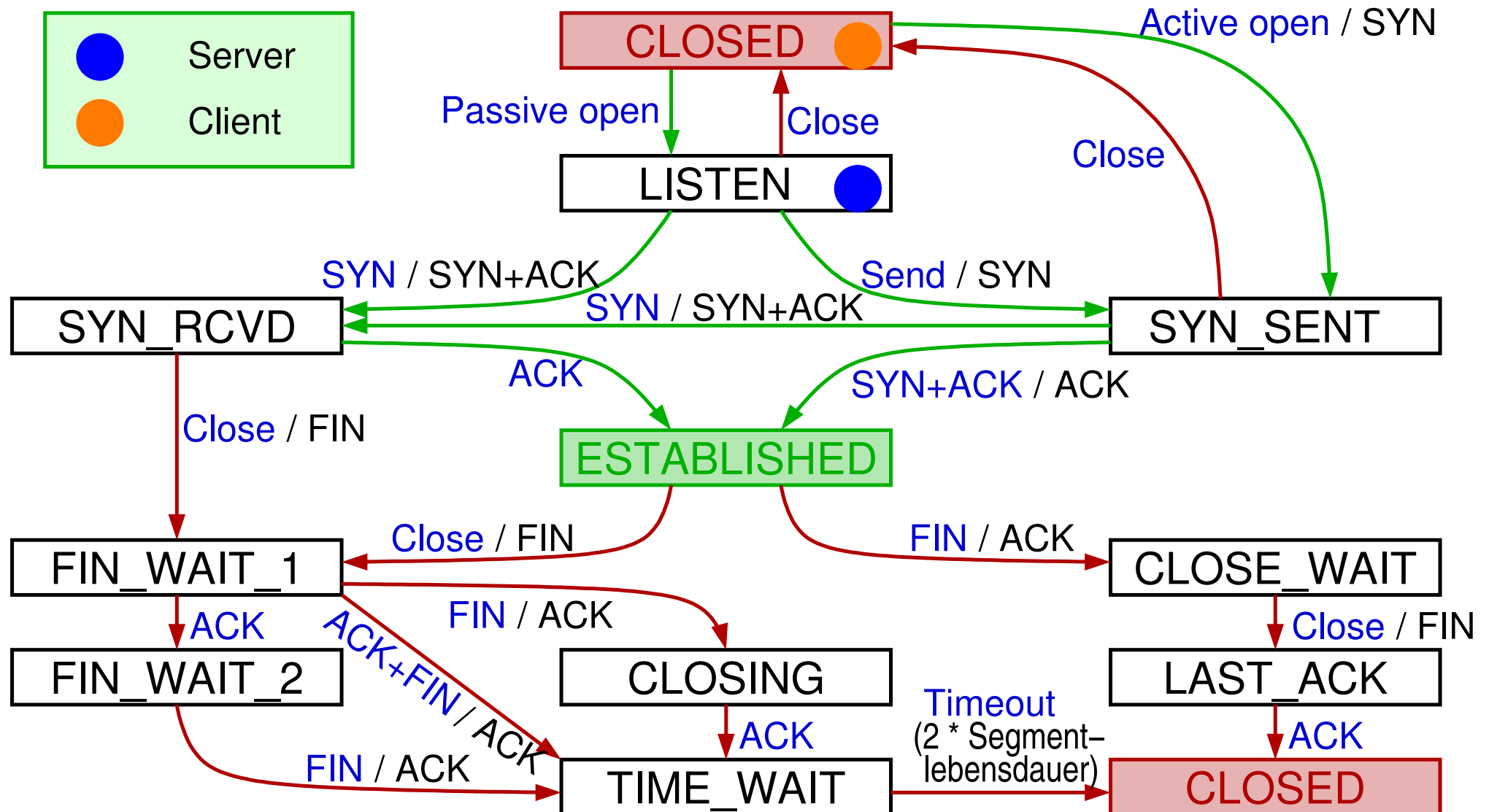
Zustände einer TCP-Verbindung



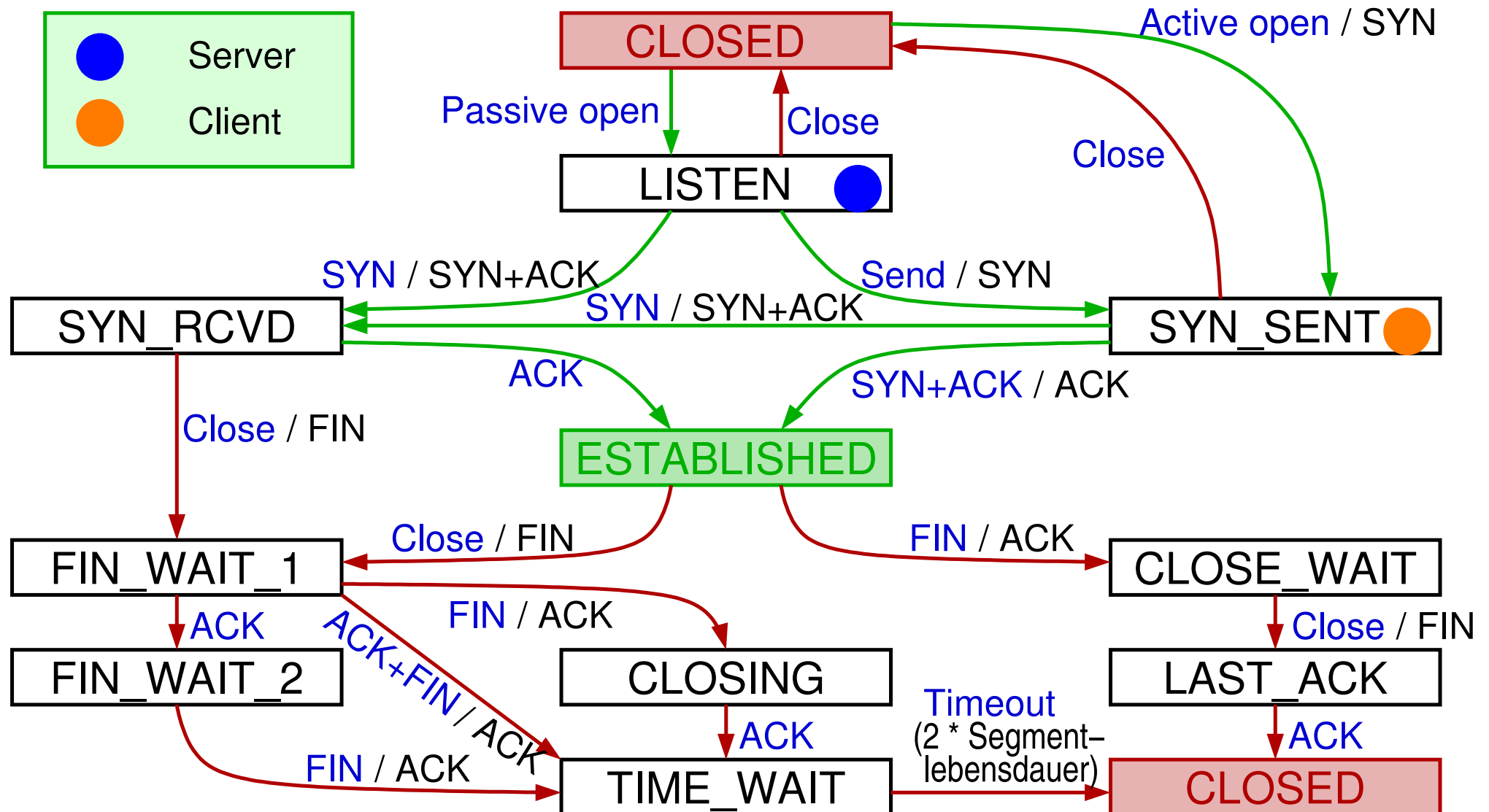
Zustände einer TCP-Verbindung ...



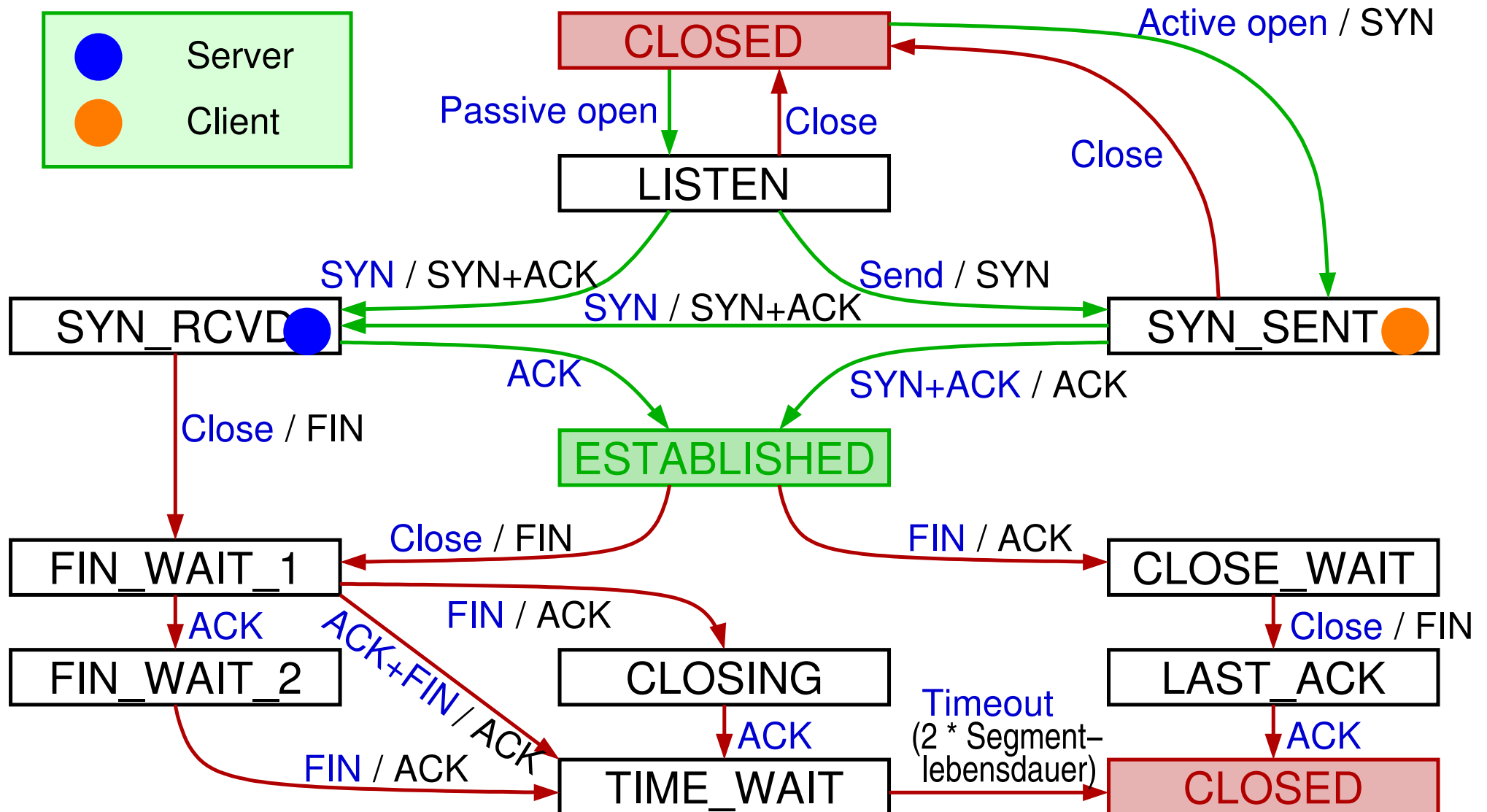
Zustände einer TCP-Verbindung ...



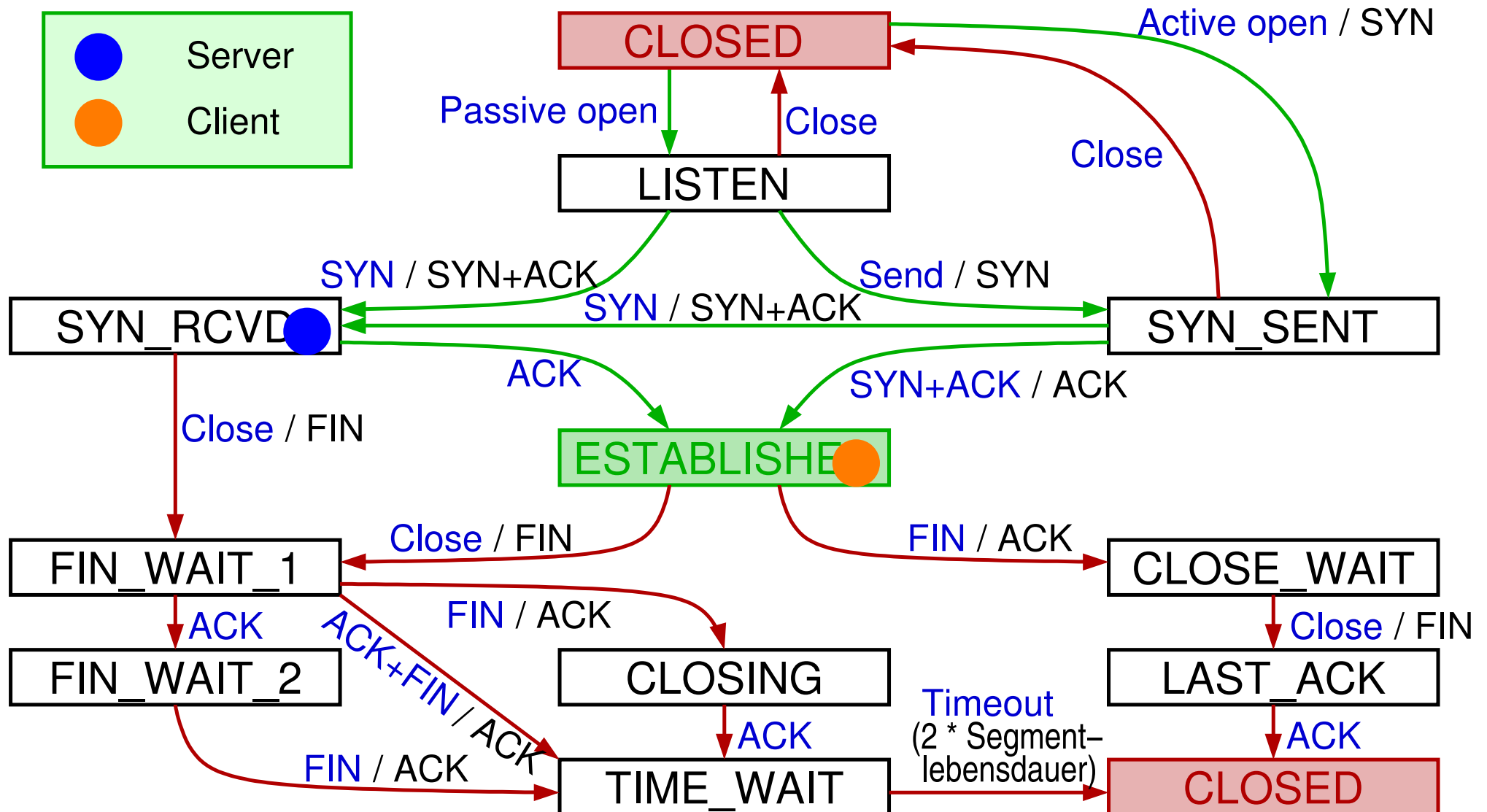
Zustände einer TCP-Verbindung ...



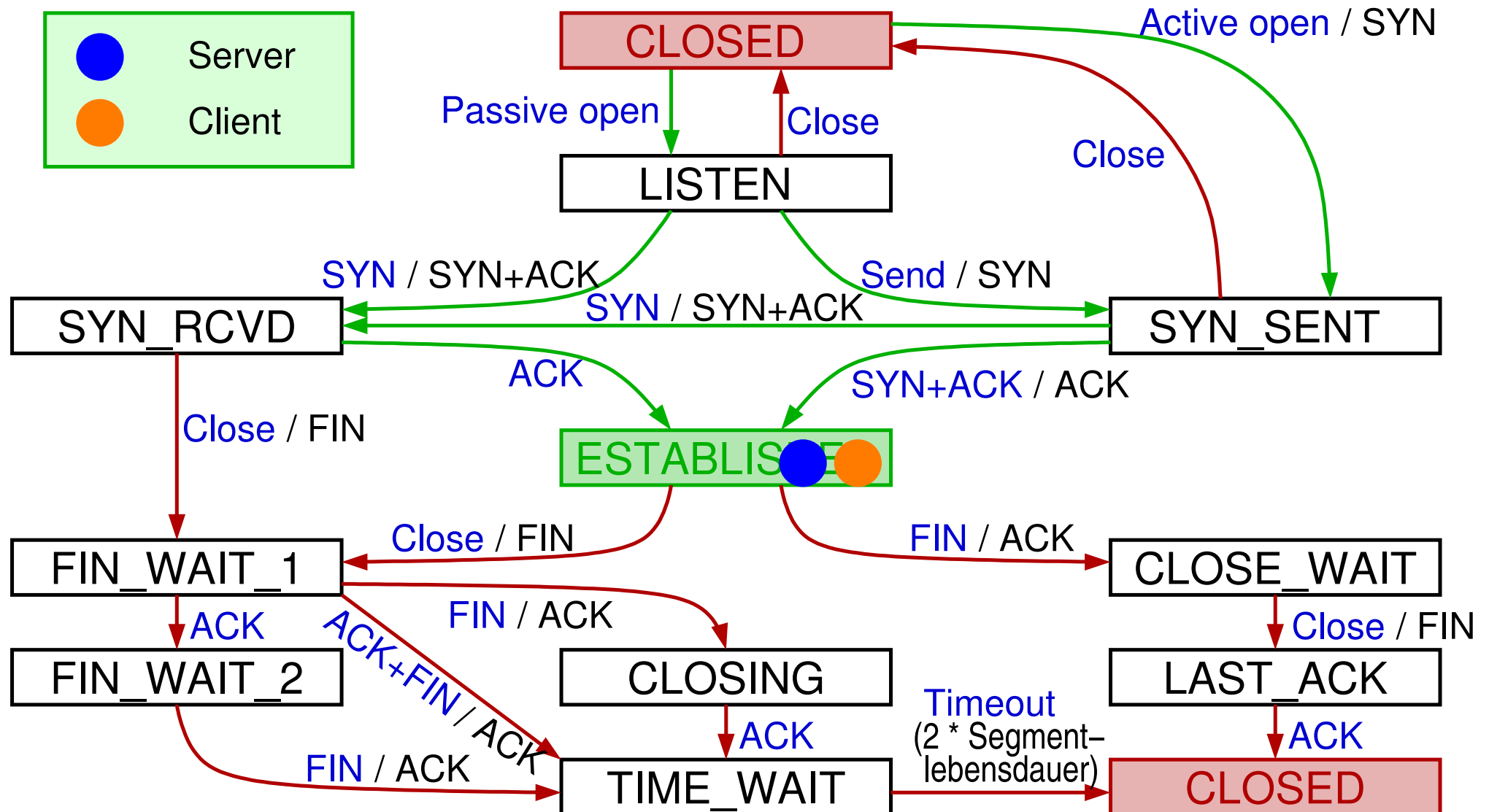
Zustände einer TCP-Verbindung ...



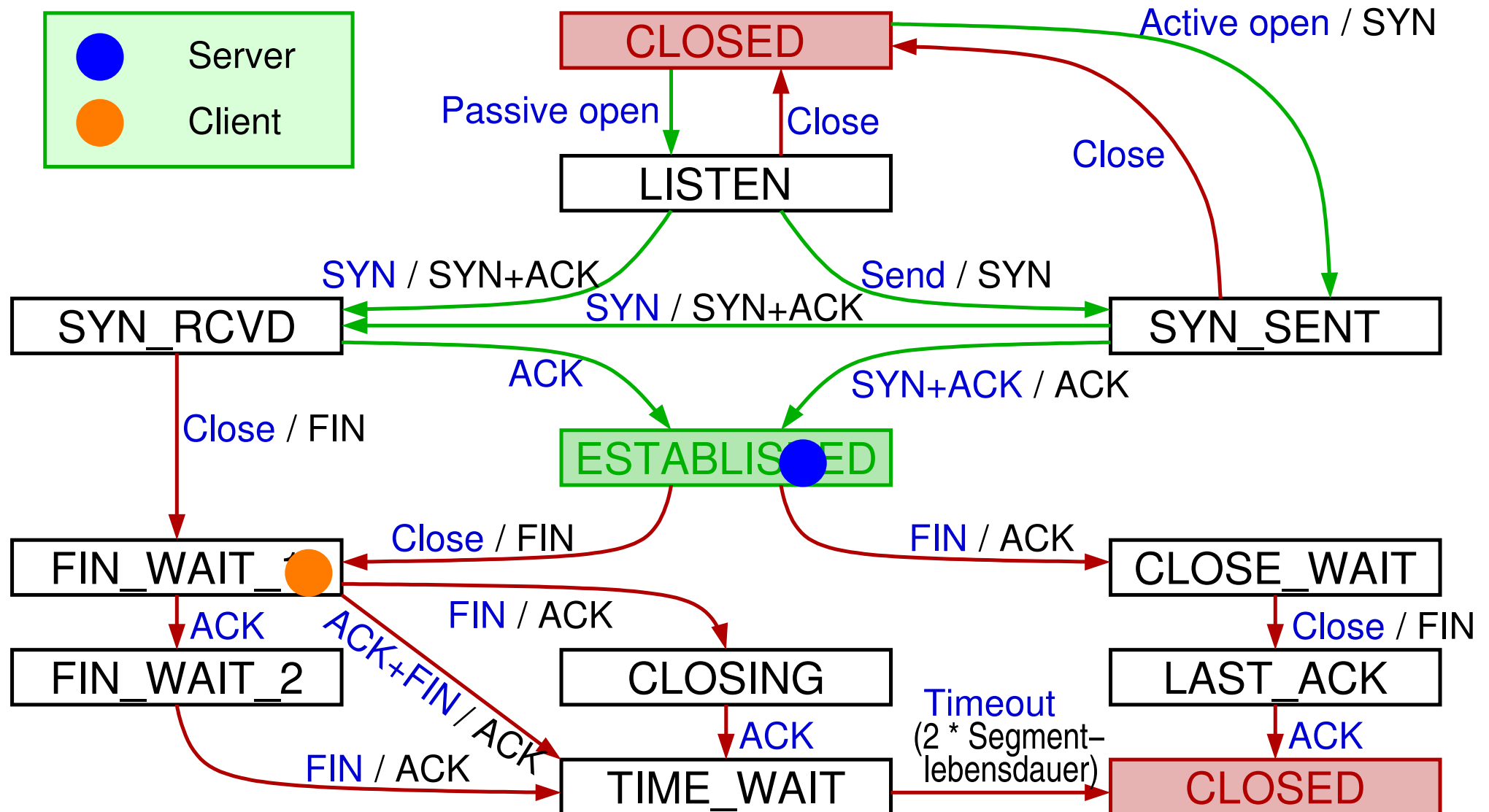
Zustände einer TCP-Verbindung ...



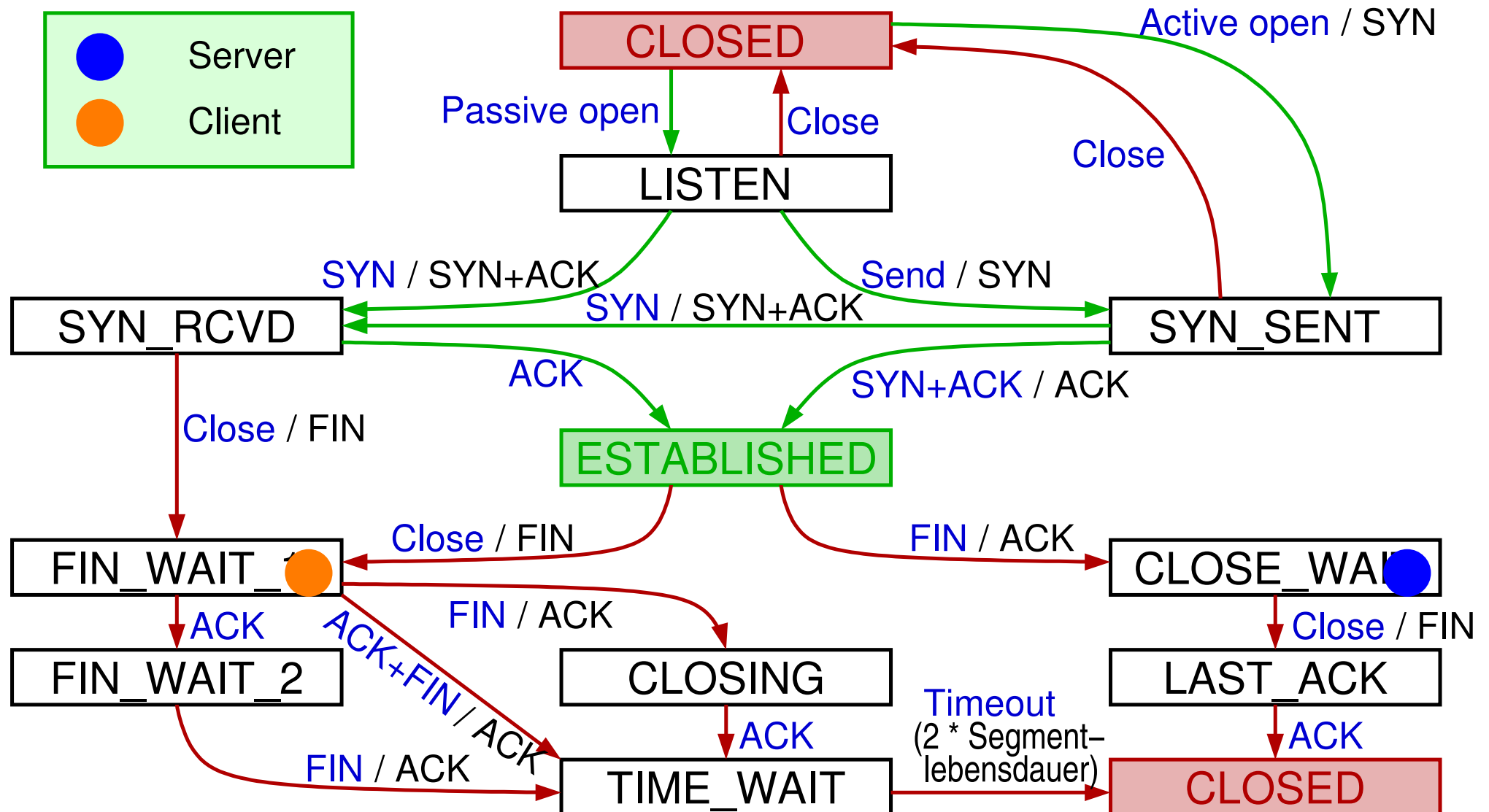
Zustände einer TCP-Verbindung ...



Zustände einer TCP-Verbindung ...



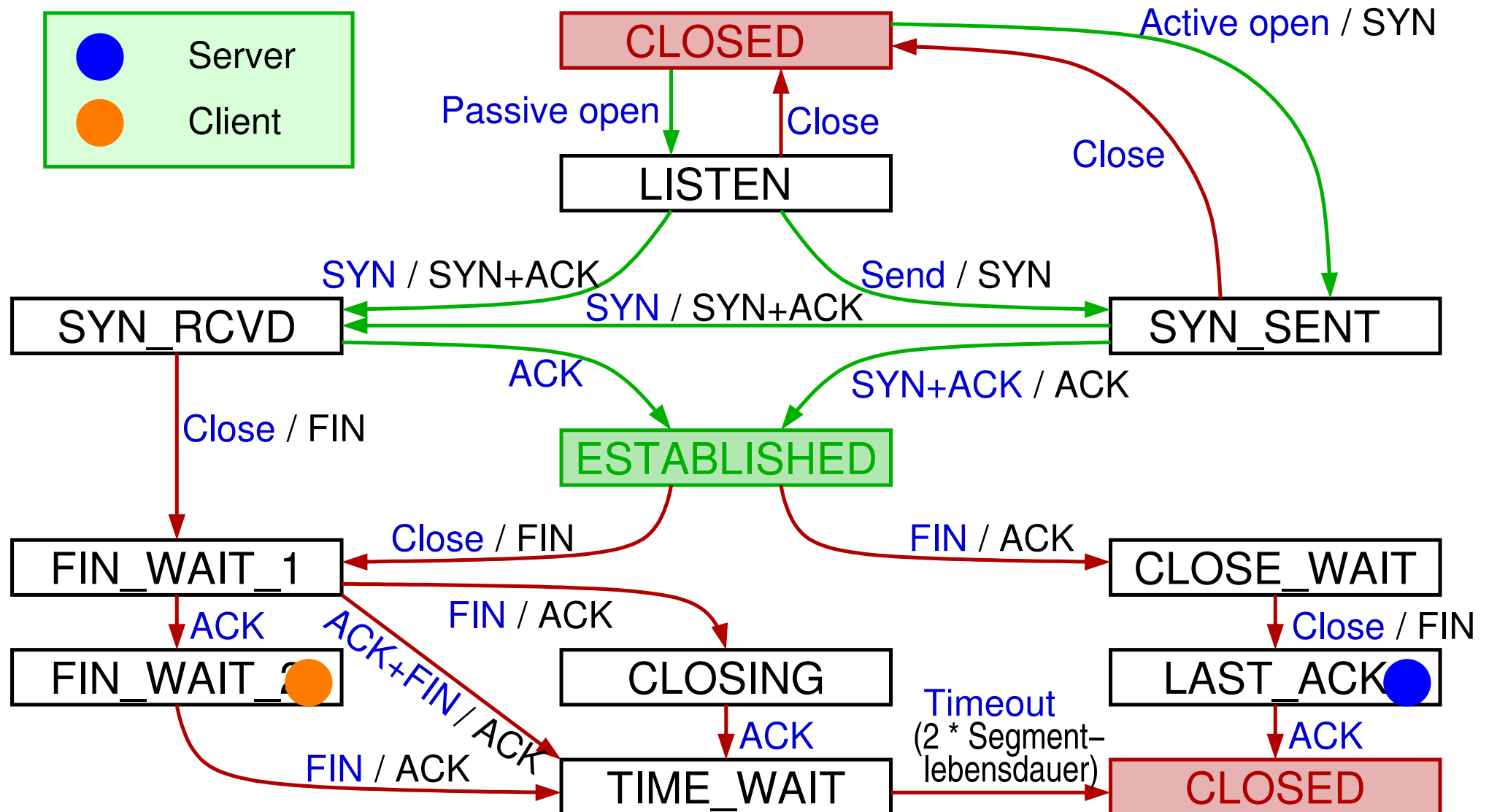
Zustände einer TCP-Verbindung ...



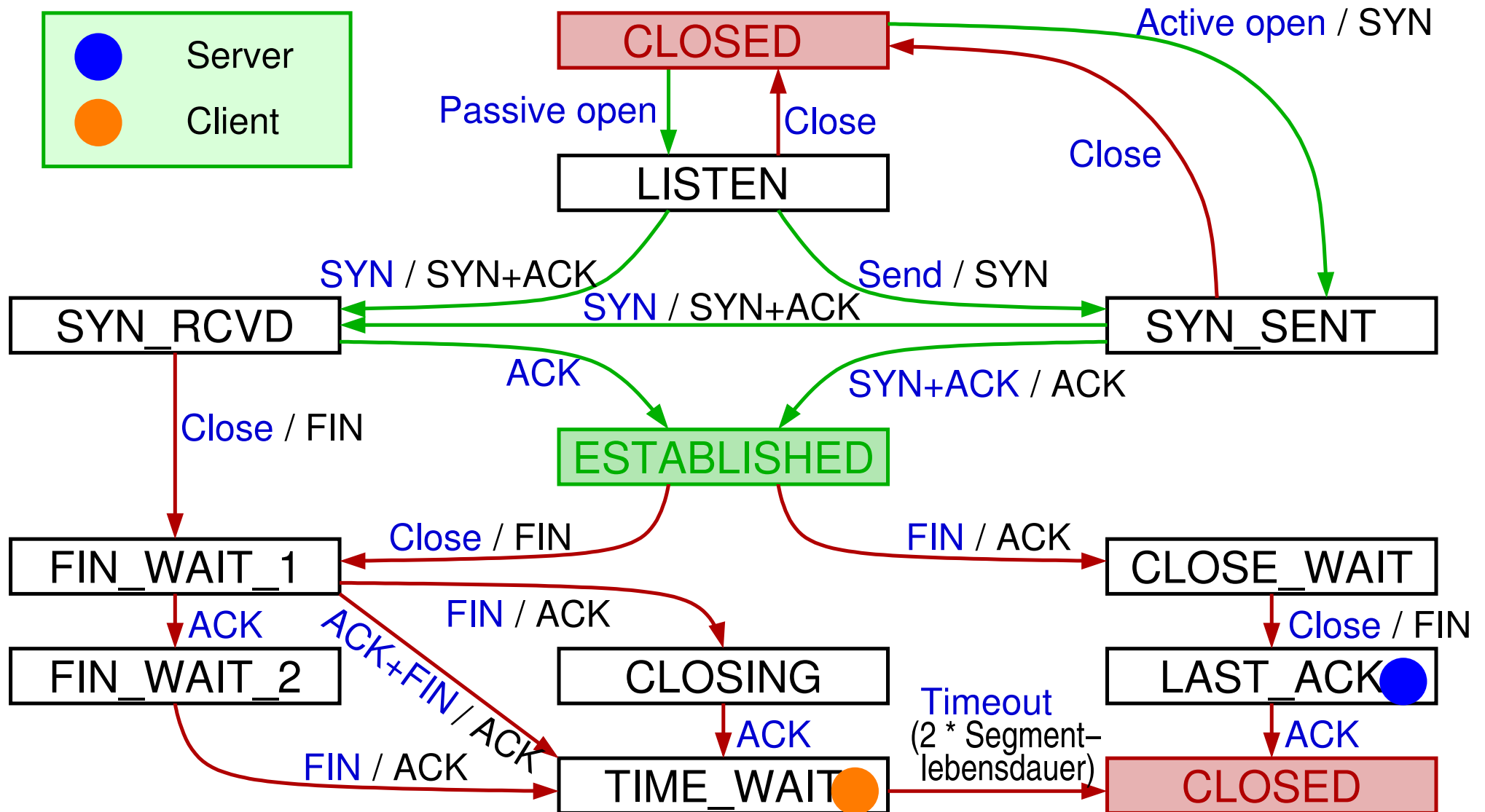
The diagram illustrates the TCP connection lifecycle with the following states and transitions:

- States:**
 - CLOSED** (Red box)
 - LISTEN** (White box)
 - SYN_RCVD** (White box)
 - SYN_SENT** (White box)
 - ESTABLISHED** (Green box)
 - FIN_WAIT_1** (White box)
 - FIN_WAIT_2** (White box)
 - CLOSING** (White box)
 - TIME_WAIT** (White box)
 - CLOSE_WAIT** (White box)
 - LAST_ACK** (White box)
- Transitions:**
 - LISTEN to CLOSED:** Passive open (Green arrow)
 - CLOSED to LISTEN:** Close (Red arrow)
 - LISTEN to SYN_SENT:** Send / SYN (Green arrow)
 - SYN_SENT to LISTEN:** Close (Red arrow)
 - SYN_SENT to SYN_RCVD:** SYN / SYN+ACK (Green arrow)
 - SYN_RCVD to SYN_SENT:** SYN / SYN+ACK (Green arrow)
 - SYN_SENT to ESTABLISHED:** SYN+ACK / ACK (Green arrow)
 - SYN_RCVD to ESTABLISHED:** ACK (Green arrow)
 - SYN_RCVD to FIN_WAIT_1:** Close / FIN (Red arrow)
 - ESTABLISHED to FIN_WAIT_1:** Close / FIN (Red arrow)
 - ESTABLISHED to CLOSE_WAIT:** FIN / ACK (Red arrow)
 - FIN_WAIT_1 to FIN_WAIT_2:** ACK (Blue arrow)
 - FIN_WAIT_1 to CLOSING:** FIN / ACK (Red arrow)
 - FIN_WAIT_2 to CLOSING:** ACK+FIN / ACK (Red arrow)
 - FIN_WAIT_2 to TIME_WAIT:** FIN / ACK (Red arrow)
 - CLOSING to TIME_WAIT:** ACK (Blue arrow)
 - CLOSE_WAIT to LAST_ACK:** Close / FIN (Red arrow)
 - LAST_ACK to CLOSED:** ACK (Blue arrow)
 - TIME_WAIT to CLOSED:** Timeout (2 * Segment-lebensdauer) (Red arrow)
 - CLOSED to CLOSED:** Active open / SYN (Green arrow)

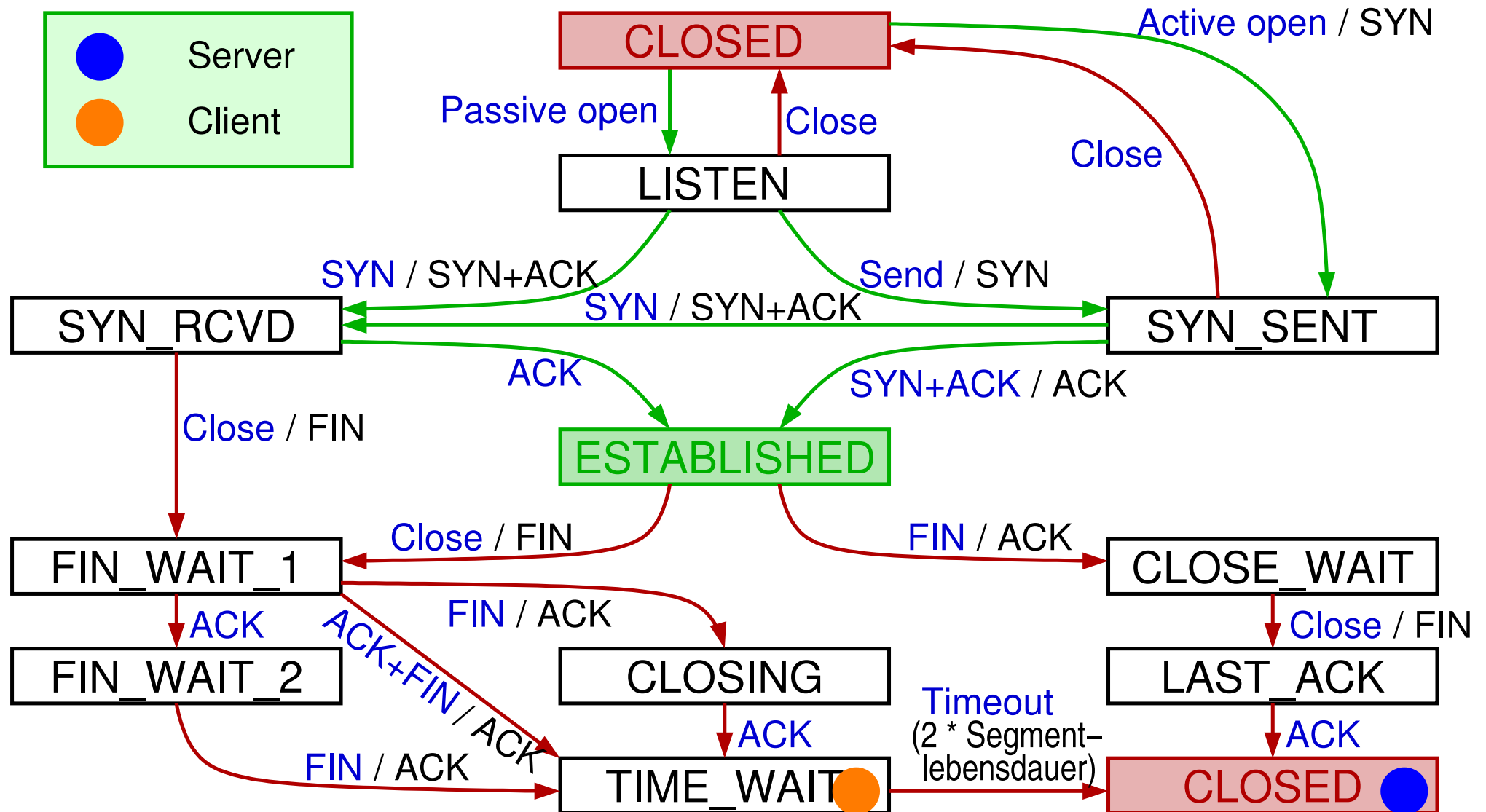
Zustände einer TCP-Verbindung ...



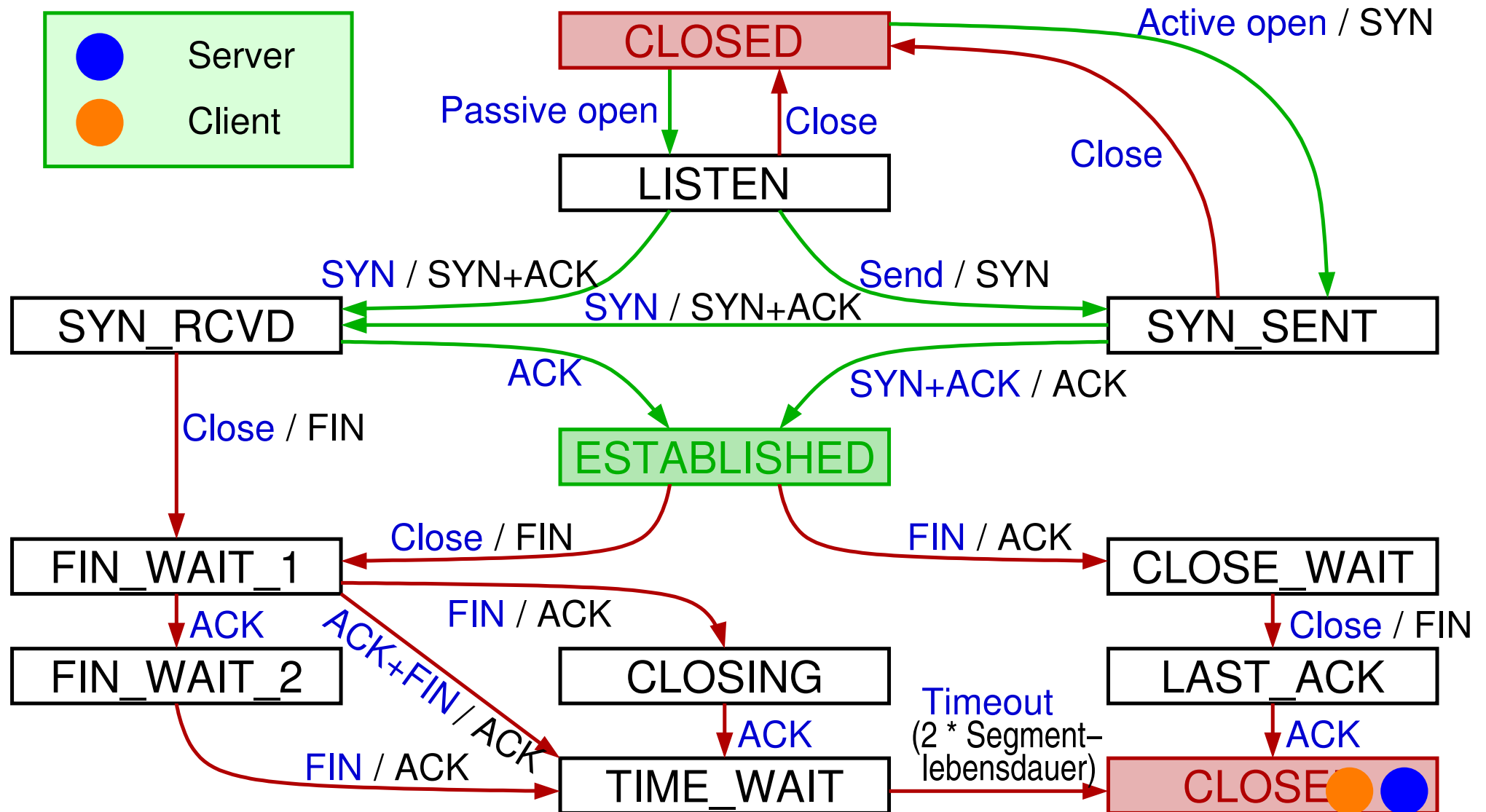
Zustände einer TCP-Verbindung ...



Zustände einer TCP-Verbindung ...



Zustände einer TCP-Verbindung ...



- ➔ Ende-zu-Ende Protokolle: Kommunikation zwischen Prozessen
- ➔ UDP: unzuverlässige Übertragung von Datagrammen
- ➔ TCP: zuverlässige Übertragung von Byte-Strömen
 - ➔ Verbindungsaufbau
- ➔ Sicherung der Übertragung allgemein
 - ➔ *Stop-and-Wait, Sliding-Window*
- ➔ Übertragungssicherung in TCP (inkl. Fluß- und Überlastkontrolle)
 - ➔ *Sliding-Window-Algorithms, adaptive Neuübertragung*

Nächste Lektion:

- ➔ Datendarstellung