Rechnernetze und verteilte Systeme

Transportschicht

Kapitel 8

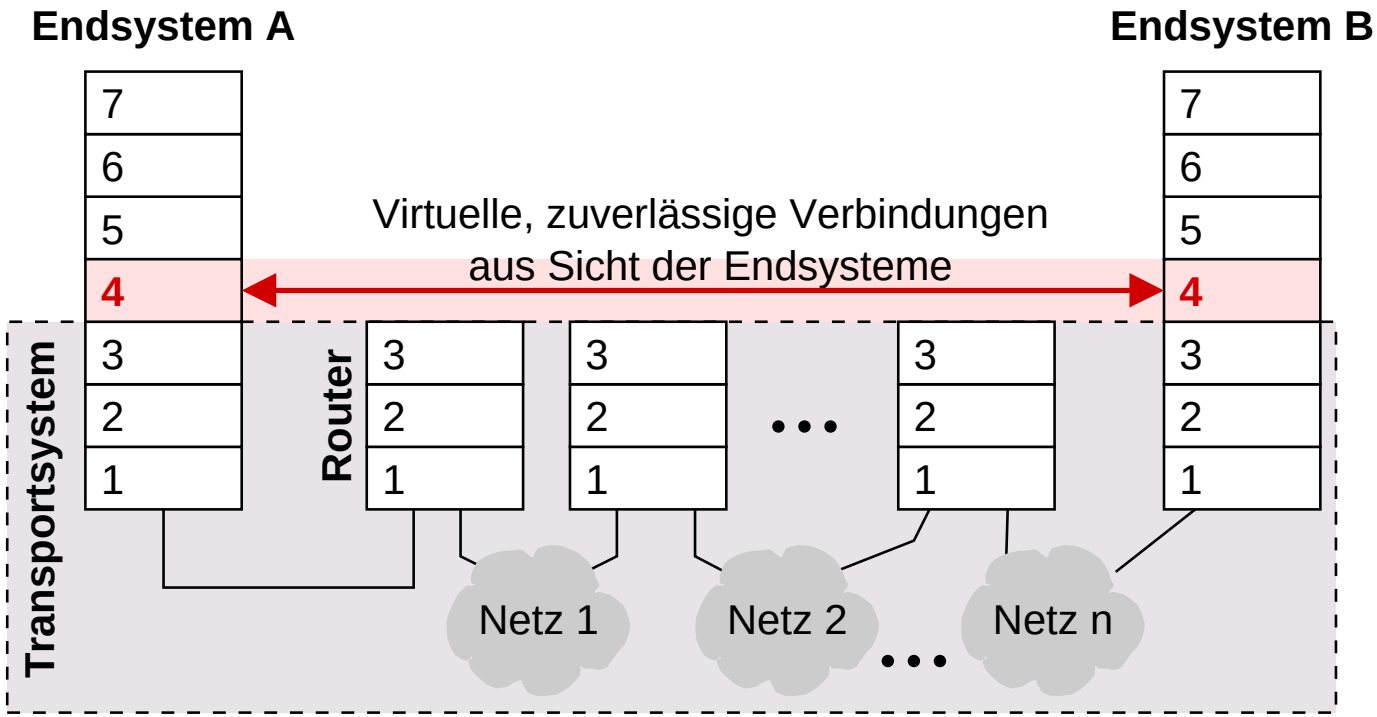
Überblick Transportschicht

- Netzunabhängiger Transport von Nachrichten zwischen Endsystemen
- Verschattung der Wege durchs Netz
- Fehlerbehandlung Ende-zu-Ende
- Anpassung der Übertragungsqualitäten, Netzauswahl

- Verbindungsloser oder -orientierter Dienst
- z.B. im Internet: TCP (verbindungsorientiert), UDP (verbindungslos)
- Splitting/Multiplex über Anwendungen/Prozesse

- Verbindungen bestehen zwischen zwei Endsystemen
  - Transitnetze bzw. Netzknoten für Transportprotokoll nicht sichtbar

  ⇒ Pfad unbekannt (bzw. irrelevant)
- Virtuelle Verbindung



Rechnernetze und verteilte Systeme
Transportschicht

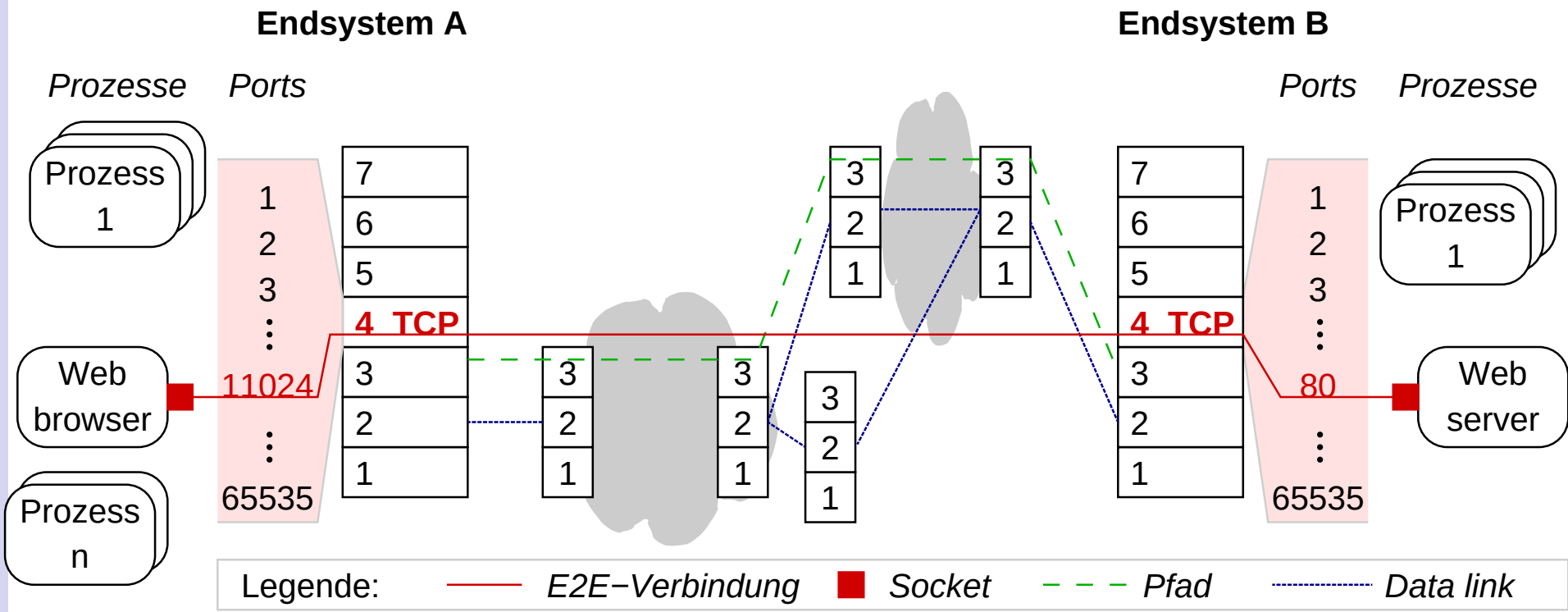
Transmission Control Protocol

Kapitel 8.2

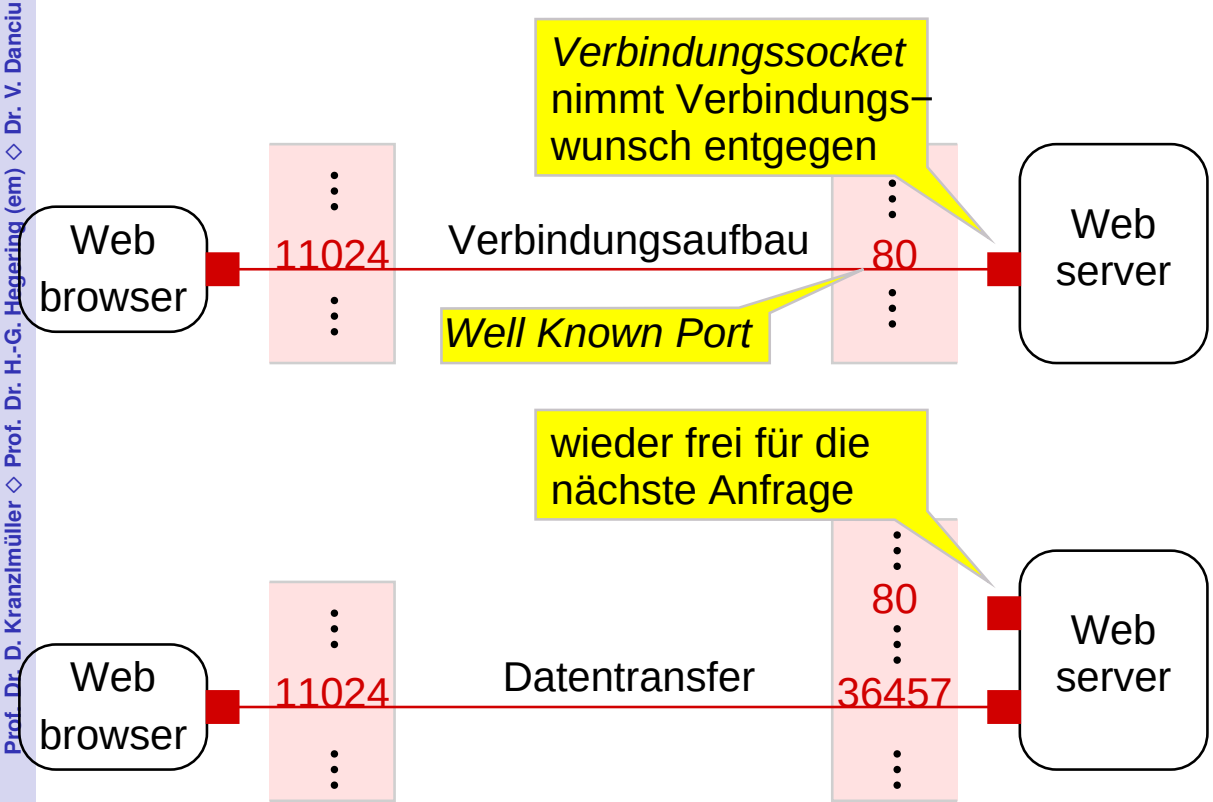
8.2 Transmission Control Protocol
Überblick

- Internet Transportprotokoll, RFC 793
- TCP unterstützt Ende-zu-Ende-Transportverbindungen (Point-to-Point)
  - vollduplex,
  - mit Fehlerbehandlung
  - mit Flusssteuerung (Überlastkontrolle)
- TCP ist Byte-Strom-orientiert, Sequenz- und Quittungsnr. beziehen sich auf Bytes
- TCP bietet VO-Dienst, Aufbau mit 3-way-handshake
- TCP-Nutzer sind über Sockets adressierbar
  - Socket: (IP-Adresse Host, lokale PortNr)
  - TCP-Verbindung=(socket1, socket2)
- TCP unterstützt ein Multiplexen von Anwendungen
- TCP kann Dienstdaten zwischenspeichern, tatsächliche Transport-Blockung kann ein TCP-Nutzer nicht sehen

- Ende-zu-Ende-Verbindung an Dienst bzw. Applikation gebunden
- Socket: Endpunkt für Kommunikation
  - beschrieben durch Schicht-3-Adresse und Portnummer
  - wird erzeugt von und „gehört“ einem Prozess
  - bietet Schnittstelle für zuverlässige Byte-Übertragung
- Verbindung gegeben durch ein Socket-Paar

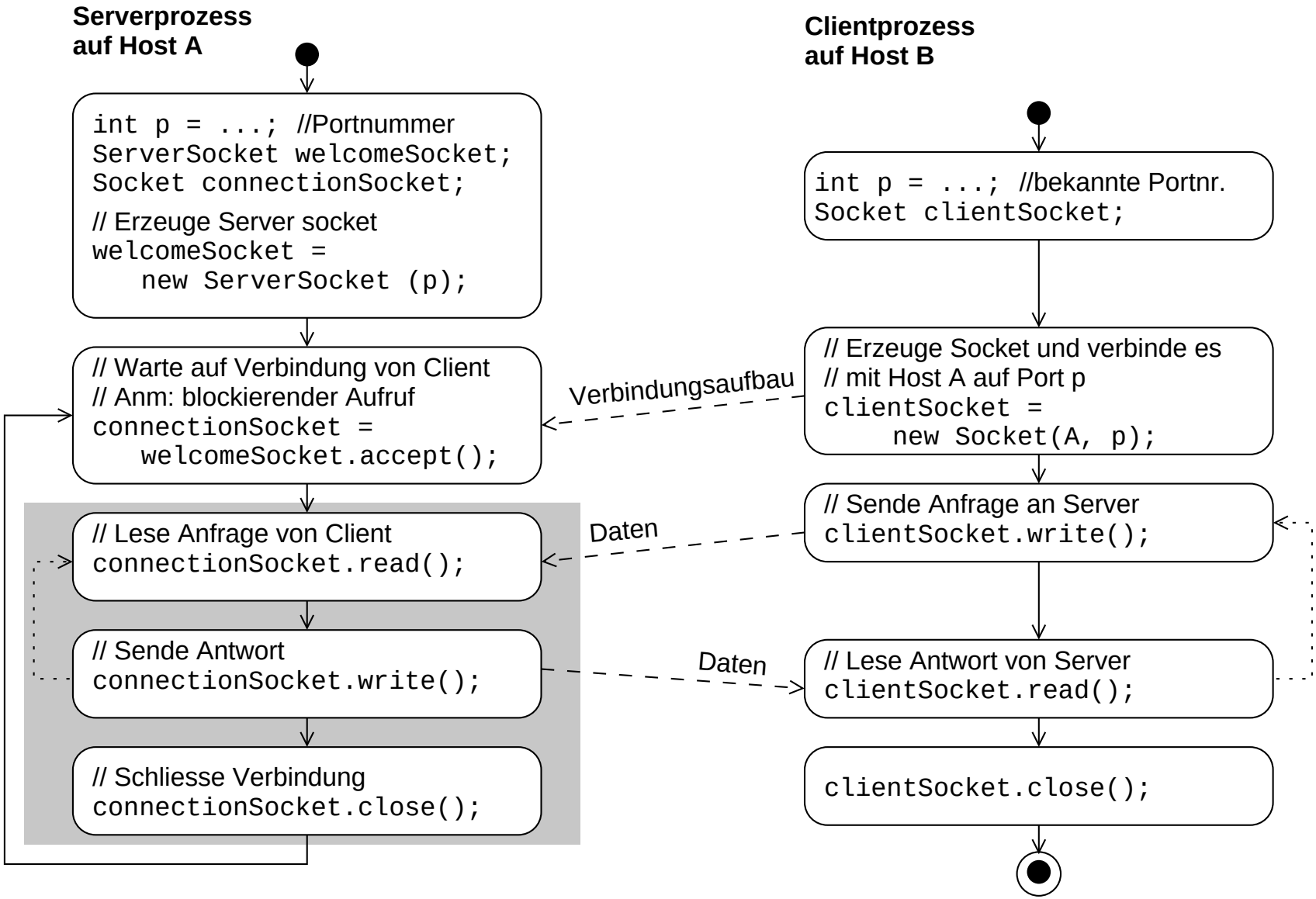


- Portnummern vergibt IANA (Internet Assigned Numbers Authority)
  - *Well Known Ports* 0 – 1023: 1-255 für TCP-Anwendungen, 256-1023 für Unix-Anwendungen
  - *Registered Ports* 1024 – 49151: meist herstellerspezifisch
  - *Dynamic and/or Private Ports* 49152 – 65535: frei nutzbar
  - Unix-Systeme: Abbildung Port/Dienst in Datei `/etc/services`



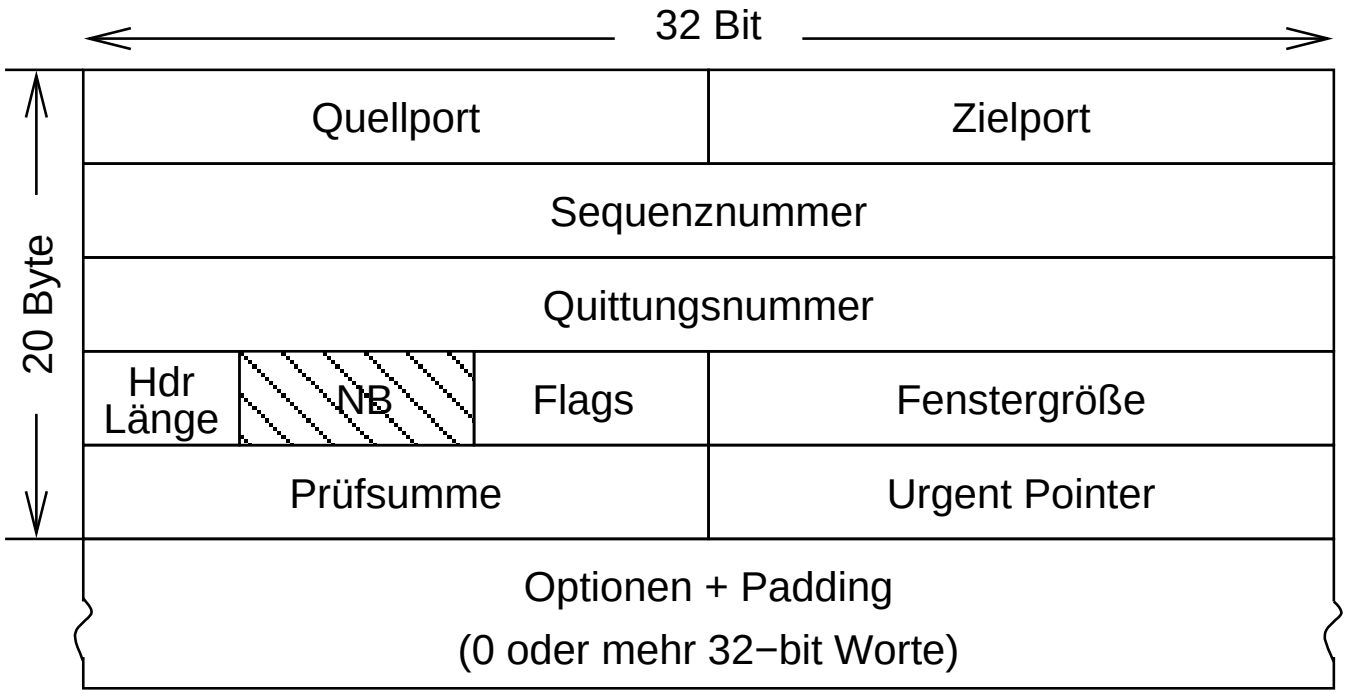
Well Known Ports (Bsp.)	
21	FTP
22	Secure Shell
23	telnet
25	SMTP
143	IMAP
Registered Ports (Bsp.)	
2628	DICT
2481	Oracle GIOP
6000	X Windows

- Socket API in BSD 4.1 UNIX, 1981 eingeführt
- Typen von Sockets:
  - TCP-Socket: bietet zuverlässige Übertragung zum korrespondierenden Client- oder Serverprozess
  - UDP-Socket: Datagrammdienst (nicht zuverlässig)
- Socket aus Programmierersicht:
  - Schnittstelle zwischen Anwendung und Transportprotokoll
  - Client-Server-Paradigma
  - Umgang mit Socket ähnlich zu dem mit einer Datei
    - Socket wird erzeugt/geöffnet
    - es wird zum Socket geschrieben / vom Socket gelesen
    - Socket wird geschlossen/zerstört
  - „Server“-Socket: wartet auf Verbindungen von Clients
  - „normales“ Socket: wird nach Verbindungsaufbau von sowohl Client- als auch Serverprozessen genutzt
  - Server/Daemonen: Erzeugung eines neuen Threads/Prozesses zur Abwicklung der Kommunikation über ein gerade verbundenes Socket (→Server kann mehrere Clients gleichzeitig bedienen)

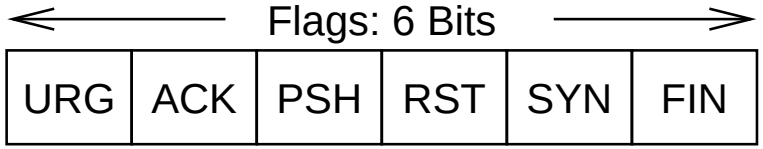

Serverprozess
auf Host A
int p = ...; //Portnummer
ServerSocket welcomeSocket;
Socket connectionSocket;
// Erzeuge Server socket
welcomeSocket =
new ServerSocket (p);
// Warte auf Verbindung von Client
// Anm: blockierender Aufruf
connectionSocket =
welcomeSocket.accept();
// Lese Anfrage von Client
connectionSocket.read();
// Sende Antwort
connectionSocket.write();
// Schliesse Verbindung
connectionSocket.close();
Clientprozess
auf Host B
int p = ...; //bekannte Portnr.
Socket clientSocket;
// Erzeuge Socket und verbinde es
// mit Host A auf Port p
clientSocket =
new Socket(A, p);
// Sende Anfrage an Server
clientSocket.write();
// Lese Antwort von Server
clientSocket.read();
clientSocket.close();
Verbindungsaufbau
Daten
Daten

8.2 Transmission Control Protocol

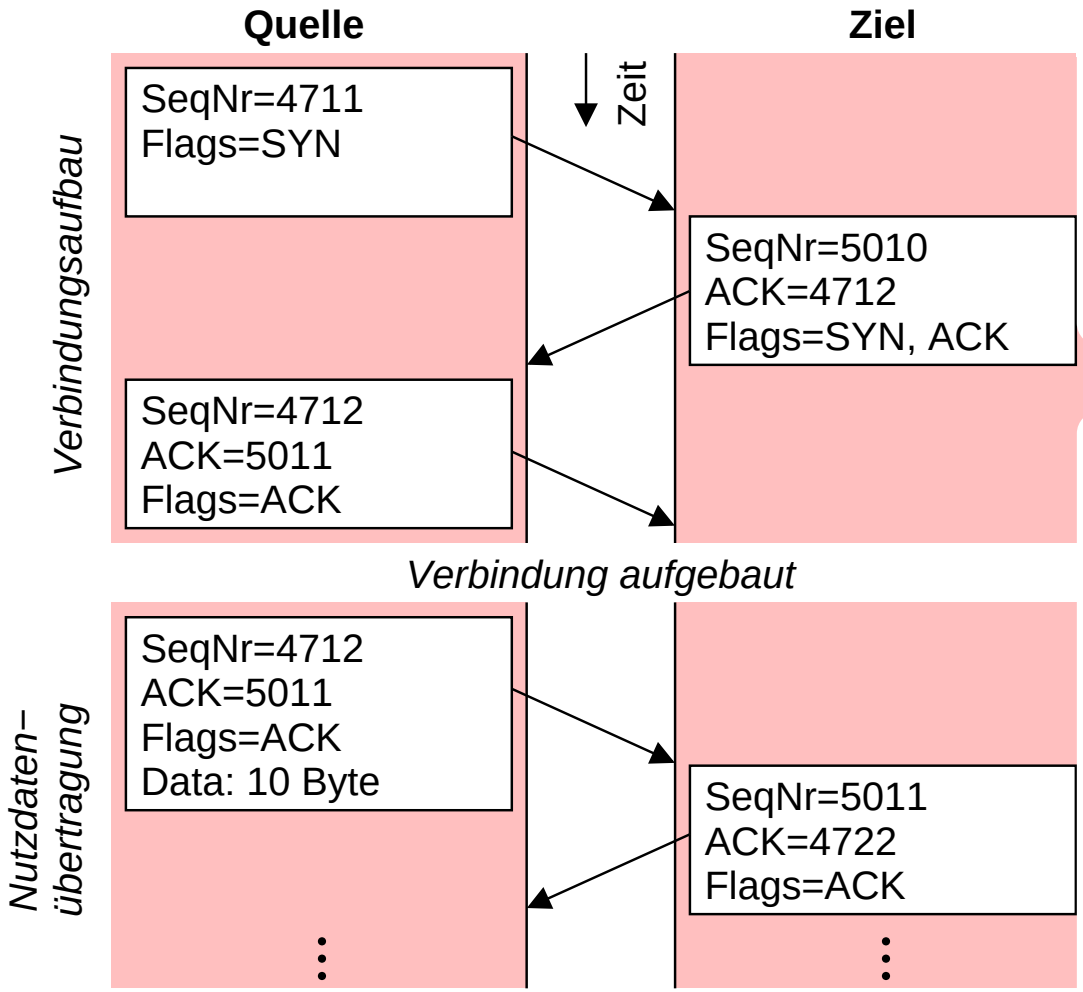
Protokoll-Header (1)



- Eine TCP-PDU wird oft auch als Segment bezeichnet
- Source Port, Destination Port →Anwendungsprozesse
- Sequenznummer
  - Senden und Empfangen (zählt auf Byte-Strom)
  - Sequenznr. eines Segments ist Bytestromnr. des ersten Bytes im Segment
- Header-Länge (4 Bit) in Anzahl 32 Bit-Wörter
- NB: nicht benutzt, 6 Bits (schraffiert)



- Flags (6 Bits)
  - URG Urgent Field benutzt (entspricht interrupt data)
  - ACK Quittungssequenznr gültig
  - PSH *pushed data* (wird sofort gesendet)
  - RST *reset connection* oder Aufbauwunsch abgelehnt
  - SYN Aufbauwunsch (SYN=1, ACK=0), Quittung dazu (SYN=1, ACK=1)
  - FIN Abbauwunsch, keine weiteren Daten
- Fenstergröße (variabel): Anzahl der Bytes die ab letzter Quittung gesendet werden dürfen
- Prüfsumme: Einerkomplement der Summe aller 16-Bit-Worte über Pseudoheader, TCP-Header und -Rumpf. Pseudoheader enthält aus dem IP-Header die Felder Quell-/Zieladresse, Protokoll u. Länge des TCP-Segments
- Urgent Pointer: zeigt auf letztes Byte in einer Kette dringlicher Daten (out of band data)
- Options: Wählbare Eigenschaften z.B. Max SegmentSize, Timestamp-Option

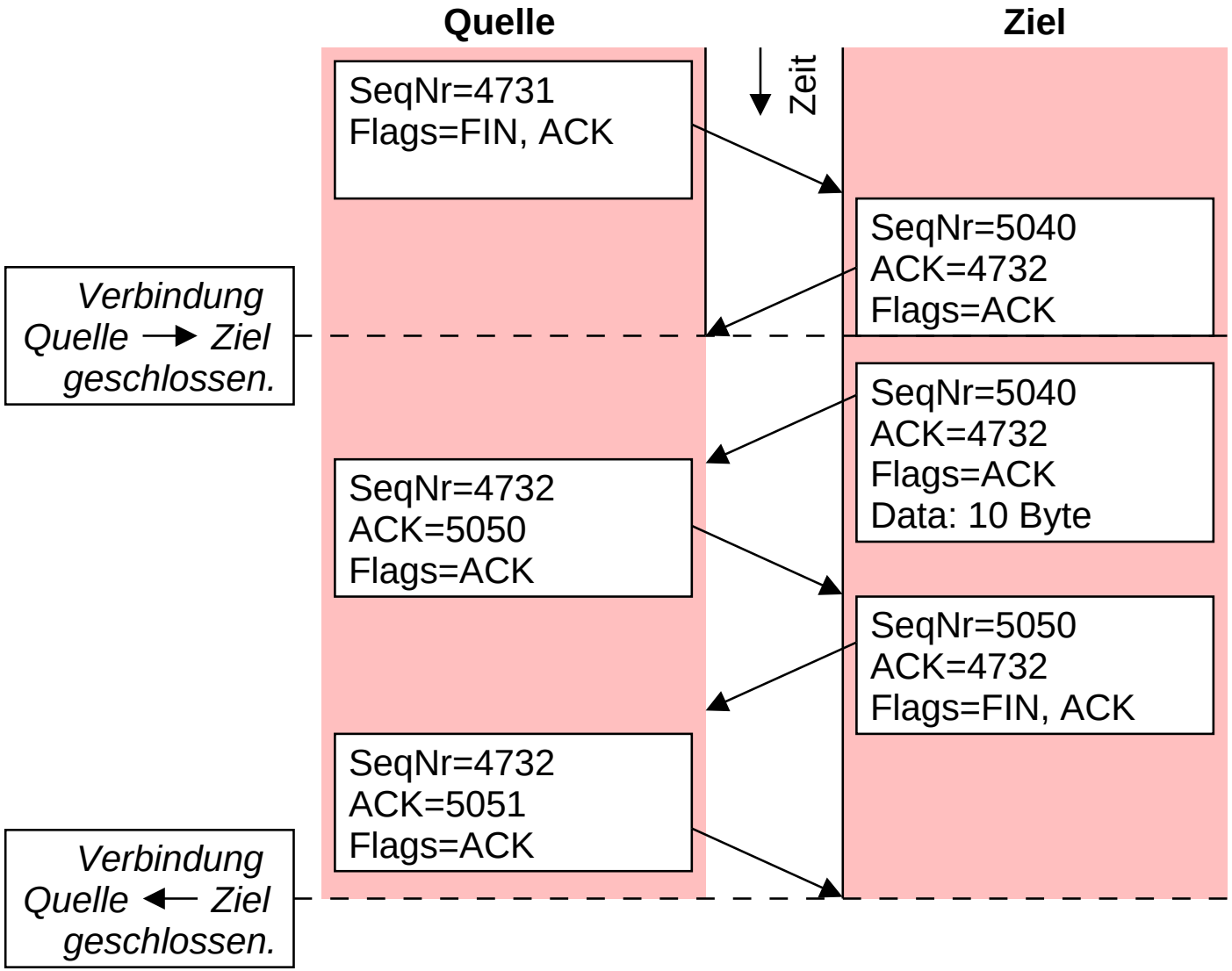


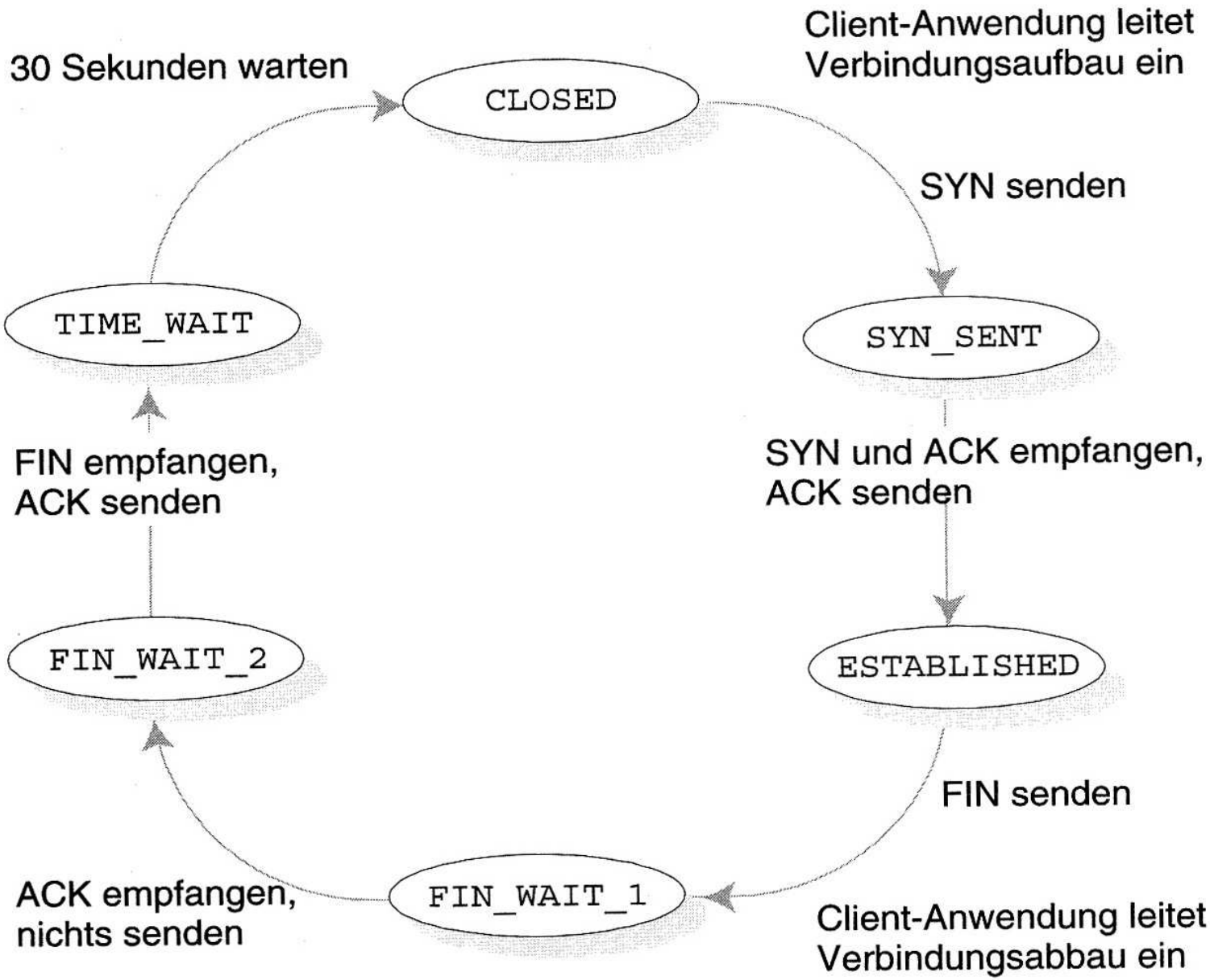
Initiale Sequenznummern

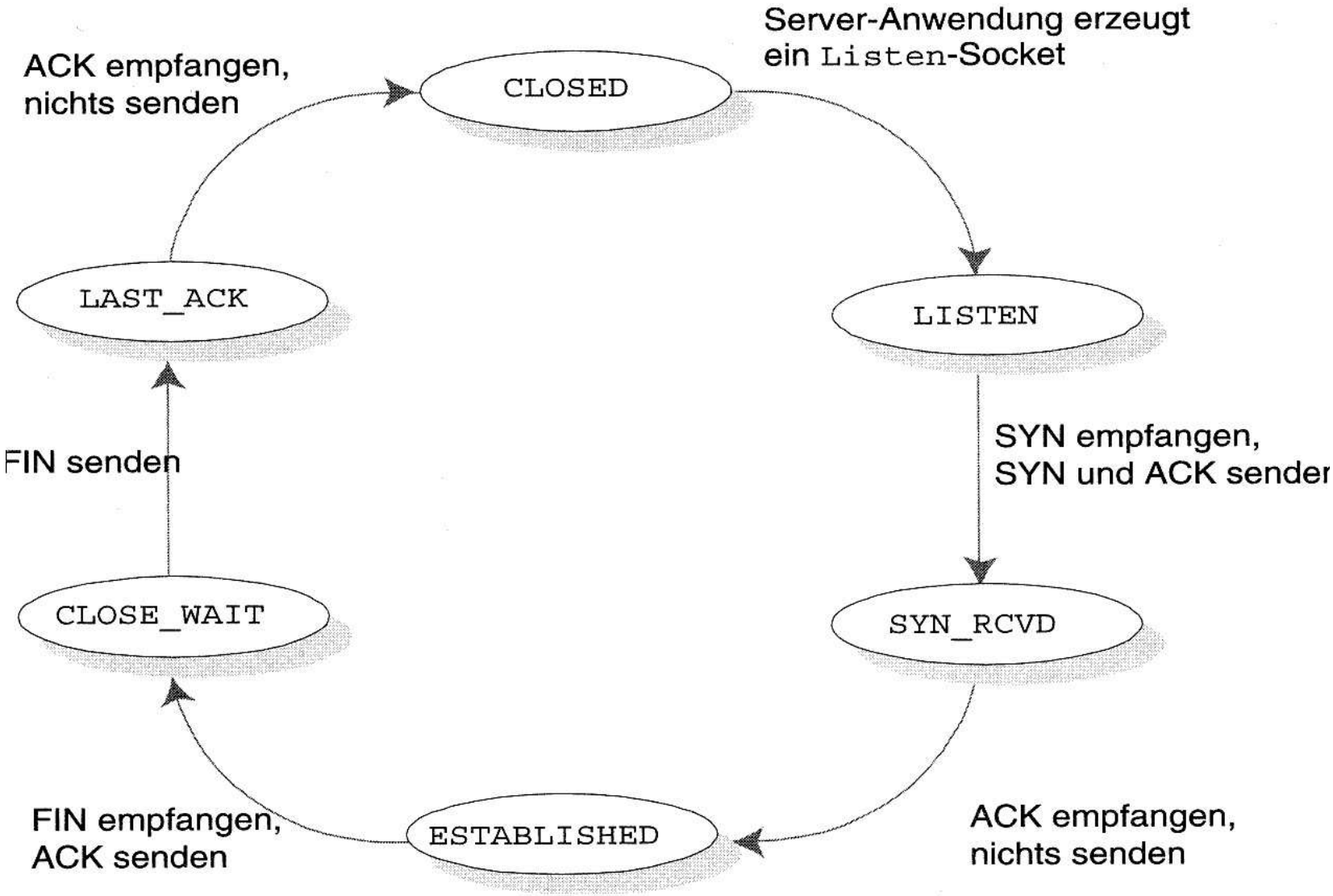
- gewählt aufgrund Zeitgeber
- Eindeutigkeit bei oft auf-/abgebauten Verbindungen

Sequenznummern werden erhöht aufgrund von

- Segmenten mit Nutzdaten (um 1 pro Byte)
- SYN oder FIN (Schutz des Verb.-Management)
- sonst: “leere” Segmente $\rightarrow$“alte” Seq.Nr. (z.B. Quittungen)

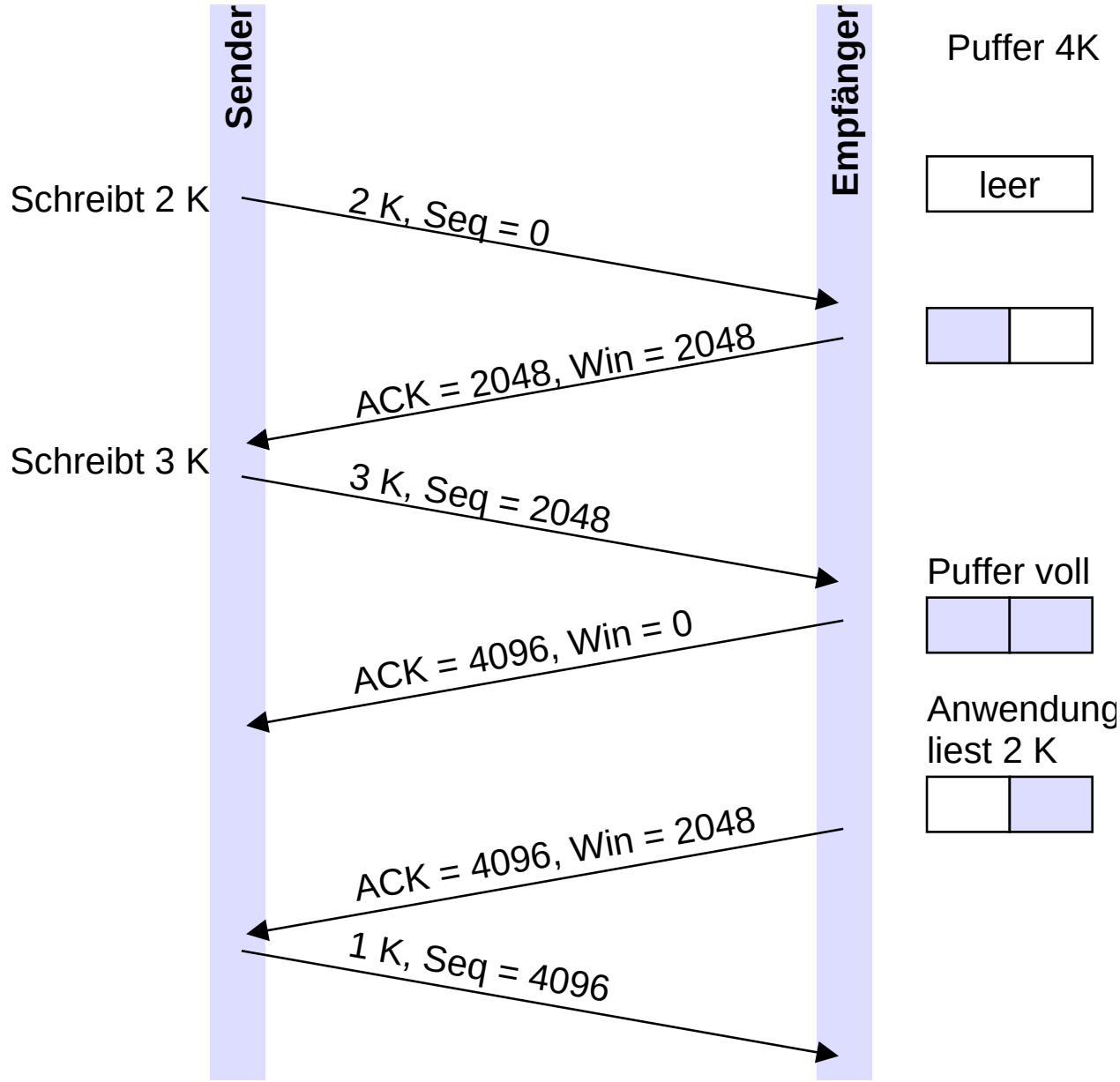

Quelle
Ziel
Zeit
SeqNr=4731
Flags=FIN, ACK
SeqNr=5040
ACK=4732
Flags=ACK
Verbindung
Quelle → Ziel
geschlossen.
SeqNr=5040
ACK=4732
Flags=ACK
Data: 10 Byte
SeqNr=4732
ACK=5050
Flags=ACK
SeqNr=5050
ACK=4732
Flags=FIN, ACK
SeqNr=4732
ACK=5051
Flags=ACK
Verbindung
Quelle ← Ziel
geschlossen.


Client-Anwendung leitet Verbindungsaufbau ein
30 Sekunden warten
CLOSED
SYN senden
TIME_WAIT
SYN_SENT
FIN empfangen, ACK senden
SYN und ACK empfangen, ACK senden
FIN_WAIT_2
ESTABLISHED
FIN senden
ACK empfangen, nichts senden
FIN_WAIT_1
Client-Anwendung leitet Verbindungsabbau ein

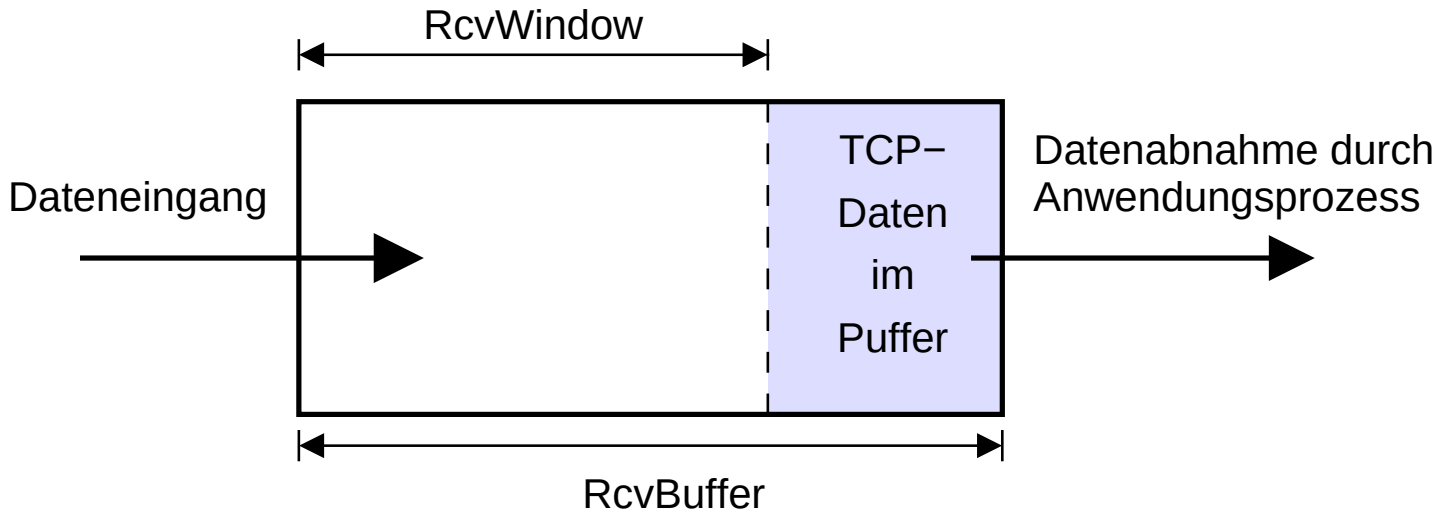

Server-Anwendung erzeugt
ein Listen-Socket
CLOSED
LISTEN
SYN empfangen,
SYN und ACK senden
SYN_RCVD
ACK empfangen,
nichts senden
ESTABLISHED
FIN empfangen,
ACK senden
CLOSE_WAIT
FIN senden
LAST_ACK
ACK empfangen,
nichts senden

8.2 Transmission Control Protocol

Flusssteuerung (1)



- Sender (A) hält Variable:
  - RcvWindow
  - LastByteSent
  - LastByteAcked
- Empfänger (B) hält:
  - RcvBuffer
  - LastByteRead: gelesen vom Anwendungsprozess in B
  - LastByteRcvd: gelesen in B von Netz für Empfangspuffer
- RcvWindow = RcvBuffer - (LastByteRcvd - LastByteRead)
- LastByteSent - LastByteAcked $\leq$ RcvWindow



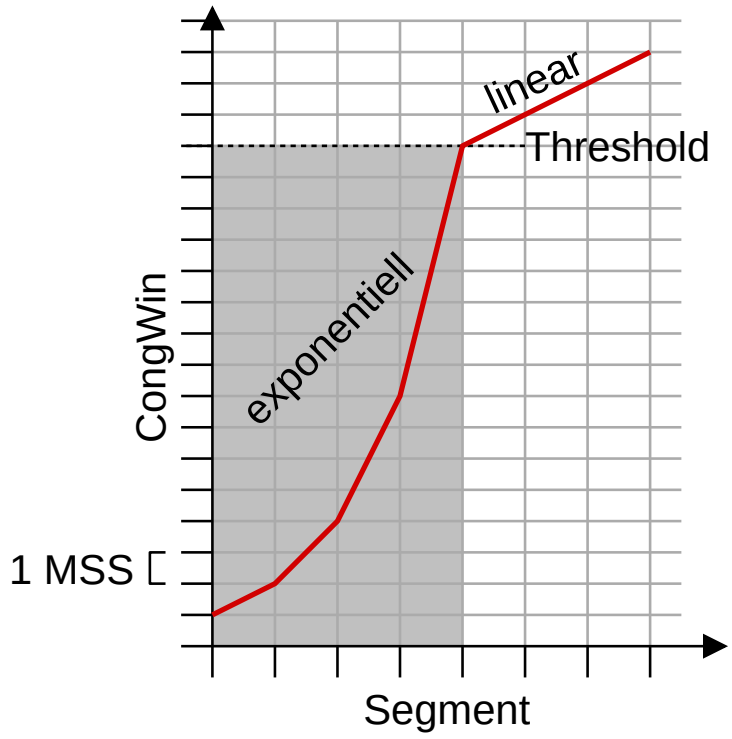
Timerfunktion	*Bedeutung*
Überwachung des Verbindungsaufbaus	Steuert die Wiederholung des Verbindungs**aufbaus** bis zum eventuellen Abbruch.
Retransmission Timer	Steuert die Wiederholung von Segmenten, die innerhalb der erwarteten Zeitspanne nicht bestätigt wurden.
Persist Timer	Zur periodischen Abfrage der aktuellen Fenstergröße eines nicht bereiten Empfängers.
Keepalive Timer	Überprüfung der Erreichbarkeit des entfernten Systems nach längeren Kommunikationspausen.
Quiet Timer	Stellt sicher, dass nach dem Neustart eines Endsystems dieses für die Dauer MSL (Maximum Segment Lifetime) keine TCP-Verbindung aufbaut. Damit wird eine unerwünschte Interaktion mit vielleicht noch bestehenden, alten Segmenten verhindert.
2MSL Timer	Wartet beim Verbindungs**abbau** das Doppelte der MSL ab, um einen möglichen Verlust des letzten ACK-Segments zu verhindern.

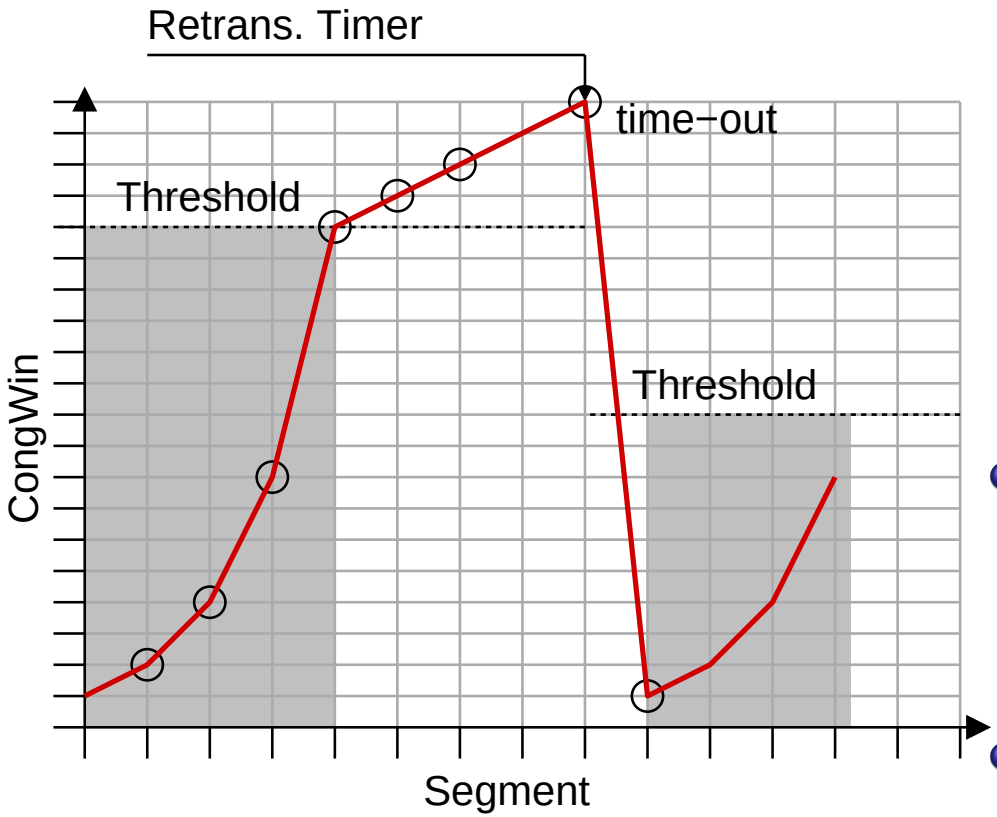
Ansatz: Einführung eines Überlastfensters (congestion window)

Jede Seite unterhält Variable:

MSS	(MaximumSegmentSize): max. Datenmenge pro Segment
CongWin	(Überlastfenster): steuert Einspeisung ins Netz: $LastByteSent - LastByteAcked \leq min(CongWin, RcvWin)$
Threshold	obere Schranke für schnelles Wachstum von CongWin

- Wichtig: Retransmission Timer (bzgl. nicht bestätigter Segmente)
- Effektives Fenster: Minimum aus Empfangsfenster und Überlastfenster
- *Slow Start*: Fenstergröße zu Anfang klein wählen
- *Congestion Avoidance*: Timerüberläufe führen zur Reduktion der Senderate





- **Slow-Start Phase**
  - Beginn: CongWin=MSS
  - Segment wird vor dessen Timerablauf bestätigt →Verdoppelung von CongWin, solange $CongWin < Threshold$
  - Threshold erreicht ⇒Ende *Slow-Start-Phase*
- **Congestion Avoidance Phase**: CongWin wächst alle RTD linear weiter, solange Quittung vor Timeout eintrifft
- **Timeout** →Zurücksetzen von Threshold auf 0,5 CongWin und CongWin=1 MSS

- Verbesserung im *Vegas*-Algorithmus, seit 1998 auch *Reno*-Algorithmus
- TCP-Übertragungsrate (mittlerer Durchsatz): $0,75 \cdot W \cdot \frac{MSS}{RTD}$, wobei W max. Überlastfenster vor Verlustsituation.

Piggy-backing von Quittungen: Summenquittung über mehrere Segmente

Path MTU Discovery Bestimmung kleinster MTU auf Pfad

$\rightarrow$ Anpassung der Segmentgröße

$\Rightarrow$ Verhinderung von IP-Fragmentierung

Nagle-Algorithmus: *blocking* zur Vermeidung zu kurzer TCP-Segmente

- Problem: Datenübergabe an TCP in kleinen Portionen

$\Rightarrow$ viele, kleine Segmente (großer Overhead)

- Lösung: Warten, bis „genug" Daten, dann geblockt übertragen

Karn-Algorithmus: Verbesserung RTD-Schätzung

- Ziel: Anpassung des Timerwertes RTT (Round Trip Time)
- Wiederholung eines Segments $\rightarrow$RTT wird nicht aktualisiert
- Fehlgeschlagene Übertragung (d.h. Timeout) $\rightarrow$RTT wird verdoppelt

Rechnernetze und verteilte Systeme
Transportschicht

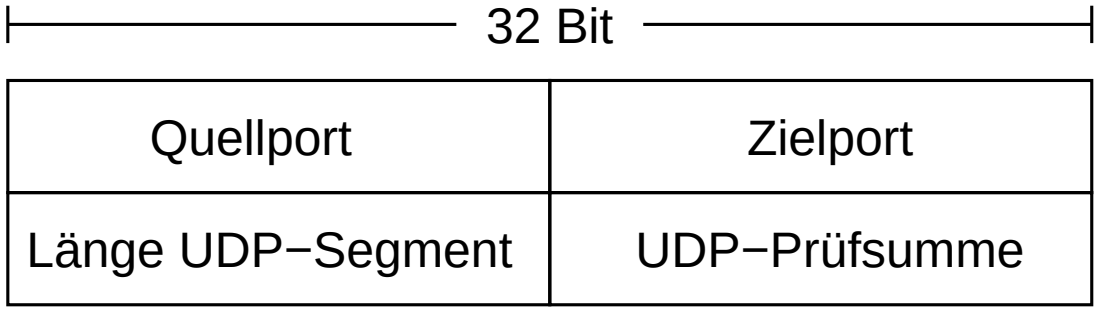
User Datagram Protocol

Kapitel 8.3

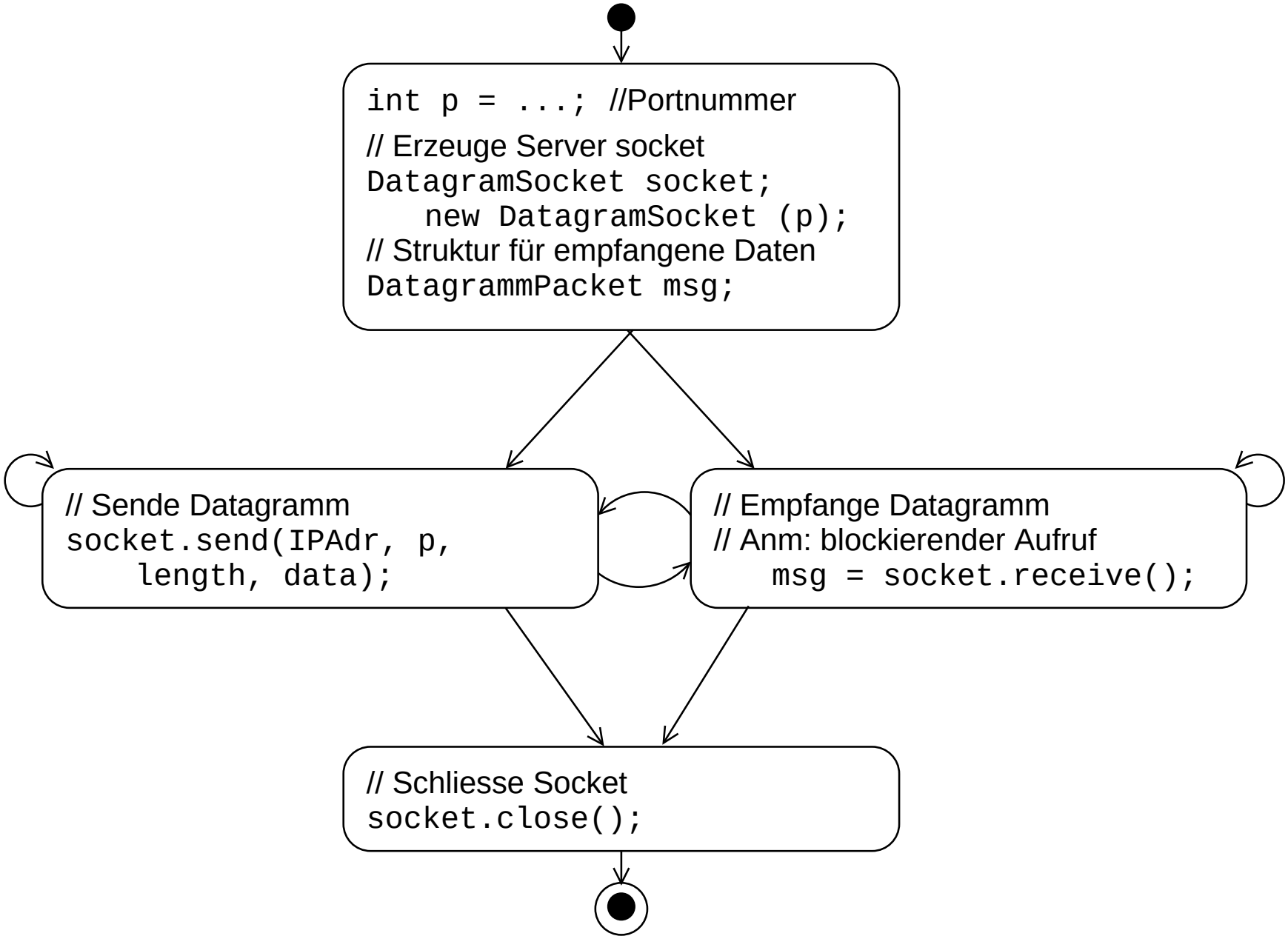
8.3 User Datagram Protocol

Überblick UDP

- verbindungsloses, unzuverlässiges Transportprotokoll
  - (kein Verbindungsaufbau, kein Verb.-Status, unregulierte Senderate)
- Multiplex von Anwendungen mittels Ports (wie TCP)
- TFTP, DNS, RPC, SNMP werden z.B. über UDP abgewickelt
- Header festgelegt in RFC 768
  - (nur 8 Byte im Gegensatz zu 20 Byte bei TCP)



- Programmieren mit UDP-Sockets
  - Senden/emfangen isolierter Datagramme
  - nur Sicht auf lokales Socket
    - Senden/empfangen zu/von mehreren Hosts
    - IP-Adresse und Port des Empfängers müssen i.d.R. beim Senden eines jeden Datagramms angegeben werden
  - Reihenfolgesicherung, Behandlung von Verlust, Steuerung Senderate: obliegen Anwendungsprogramm


int p = ...; //Portnummer
// Erzeuge Server socket
DatagramSocket socket;
new DatagramSocket (p);
// Struktur für empfangene Daten
DatagrammPacket msg;
// Sende Datagramm
socket.send(IPAdr, p,
length, data);
// Empfange Datagramm
// Anm: blockierender Aufruf
msg = socket.receive();
// Schliesse Socket
socket.close();

Fragen zu Kapitel 8

- Wann ist es u.U. sinnvoll, eine Anwendung über UDP, statt über TCP zu betreiben?
- Wie sichert man einer Anwendung einen zuverlässigen Datentransfer, auch wenn sie über UDP läuft?
- A sendet 2 TCP-Segmente an B. Das erste habe die Sequenznr. 90, das zweite 110.
  - Wie viele Daten enthält das erste Segment? Wie lang ist es insgesamt?
  - Wenn z.B. das erste Segment verloren geht, das zweite aber bei B ankommt, wie lautet die Sequenznummer in der Bestätigung von B nach A?
- Welche Mechanismen bietet TCP für eine zuverlässige Ende-zu-Ende Verbindung?
- Welches Transportprotokoll unterstützt ein Multiplexen von Anwendungen?
- TCP erledigt Flusssteuerung über „Credits“ statt über Fenstertechnik. Diskutieren Sie Vor- und Nachteile.
- Die Fragmentierung von Datagrammen und das Zusammensetzen wird von IP gemacht und ist für TCP unsichtbar. Heißt das, das sich TCP über die Reihenfolgesicherung von IP-Paketen keine Gedanken machen muss?