Lösung 11: TCP Congestion Control und Wireshark-Analyse, SSH Tunnel

Aufgabe 1: TCP Congestion Control

a) [0; 6] und [23;26], exponentielles Wachstum von cwnd; jedes korrekt bestätigte Paket führt zur Vergrößerung von cwnd.

b) [6; 16] und [17;22]; lineares Wachstum von cwnd; erst nach jeder „Runde“ wird cwnd um 1 vergrößert.

c) Durch 3 ACK-Duplikate. Bei einem Timeout wäre cwnd auf 1 gesetzt worden.

d) Durch Timeout, da cwnd auf 1 gesetzt wird.

e) *ssthres* hat den Wert 32 (ca.), da bei diesem Wert von der Slow Start Phase zur Congestion Avoidance Phase gewechselt wurde. .

f) Der bisherige Wert von *ssthres* wird in Runde 16 verringert. Da *cwnd* zu diesem Zeitpunkt 42 ist, wird *ssthres* auf 21 gesetzt (halber cwnd-Wert).
Hinweis: In Runde 17/18 ändert sich daran nichts. Vielleicht wundert man sich, warum cwnd nicht auch auf 21 gesetzt wird, sondern gleich auf 24? Der Grund: Der Paketverlust wurde durch 3 Duplicate ACKs erkannt (also viermal das gleiche ACK). Man geht aus, dass nur 1 Paket verlorengegangen ist und nicht 4. Deshalb 21 + 3!

g) In Runde 22 wird ein Problem erkannt als das *cwnd*=28. Deshalb wird ssthres in der 24. Runde den Wert 14 haben. Hinweis: Nur beim ersten Slow Start einer TCP Verbindung ist ssthres fest durch die Implementierung vorgegeben.

h) Paket 1 wird in der 1. Runde gesendet, Paket 2 und 3 in der 2. Runde, Pakete 4-7 in der 3. Runde, Pakete 8-15 in der 4. Runde, Pakete 16-31 in der 5. Runde. Das Paket 17 wird also in der 5. Runde gesendet.

Aufgabe 2: TCP in Wireshark

a)

	TCP/HTTP Client	TCP/HTTP Server
IP Adresse	192.168.1.102	128.119.245.12
Portnummer	1161	80

b) Siehe Paket #1 im Trace: Seq = 0. Das SYN Flag ist auf 1 gesetzt.
Hinweis: Eigentlich startet die TCP Verbindung gar nicht mit der Sequenznummer 0. Das ist nur die relative Sequenznummer, die Wireshark herausrechnet. In Wirklichkeit wird die Sequenznummer 232129012 übertragen. Im folgenden werden immer relative Sequenznummern betrachtet, man beginnt also beim Zählen mit der 0.

c) Siehe Paket #2 im Trace: Seq = 0, Ack = 1, Syn/Ack Flag gesetzt.
Interessant: Der TCP Server bestätigt mit ACK=1 anstatt wie erwartet mit ACK=0. Der Grund warum man ACK=0 erwarten würde: Im SYN-Paket werden noch keine Nutzdaten versendet. Für den Verbindungsaufbau gilt jedoch eine Ausnahme. Das Bestätigen eines SYN-Paketes benötigt eine Sequenznummer.

d) Wireshark markiert das Paket #199 als HTTP Post Paket. Zur Übertragung des HTTP Posts wurden 122 TCP Pakete benötigt → „*122 Reassembled Segments*“

e) Man erkennt im Wireshark, dass das 1. Segment des HTTP Posts im TCP Paket #4 steht (unter „Reassembled Segments“). Im unteren „Byte“ Feld von Wiresahrk erkennt man, dass in Paket #4 tatsächlich aus das „`HTTP Post`“ steht.

f) Am besten man fertigt eine Skizze an.

	Paketnummer in Wireshark	Sequenznummer	Paketnummer des dazugehörigen ACKs
1. Segment	4	1	6
2. Segment	5	566	9
3. Segment	7	2026	12
4. Segment	8	3486	14

g) In der SYNACK Nachricht ist das *Receive Window* 5840 Bytes groß. Es wird dann schnell vergrößert bis auf 62780 Bytes, aber nie verkleinert.
Die Entwicklung des Receive Windows erkennt man schön mit einer Wireshark-Analyse: *Statistiken, TCP Stream Graph, Window Skalierung.*

h) Jede Segmentnummer kommt nur einmal vor (mit Ausnahme des Verbindungsabbaus). Die Verbindung war also stabil ohne Retransmissions.

Aufgabe 2: TCP Congestion Control

i) Flow Control verhindert eine Überlastung des Empfängers, Congestion Control verhindert eine Überlastung des Netzwerks.

j) [0; 6] und [23;26], exponentielles Wachstum von cwnd; jedes korrekt bestätigte Paket führt zur Vergrößerung von cwnd.

k) [6; 16] und [17;22]; lineares Wachstum von cwnd; erst nach jeder „Runde“ wird cwnd um 1 vergrößert.

l) Durch 3 ACK-Duplikate. Bei einem Timeout wäre cwnd auf 1 gesetzt worden.

m) Durch Timeout, da cwnd auf 1 gesetzt wird.

n) *ssthres* hat den Wert 32 (ca.), da bei diesem Wert der Slow Start aufhört und Congestion Avoidance startet.

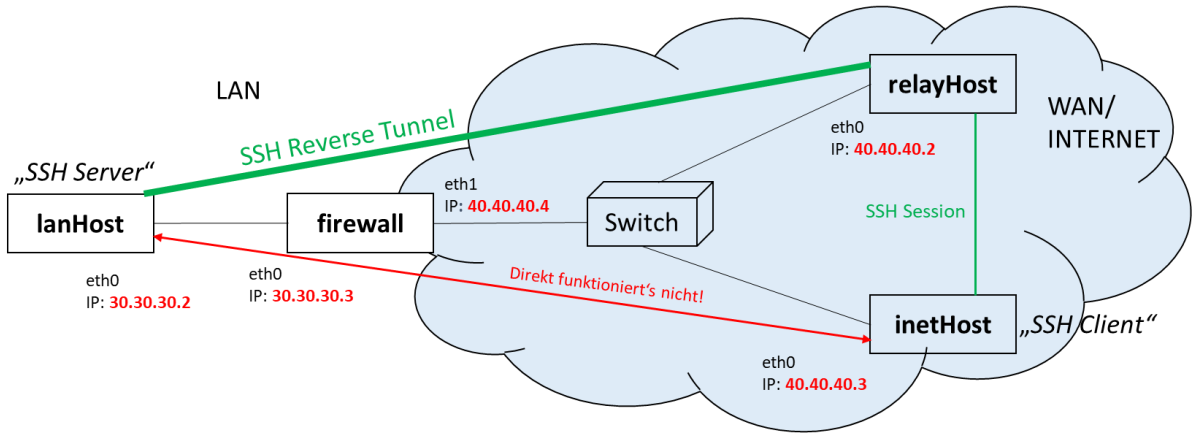
o) Der bisherige Wert von *ssthres* wird in Runde 16 verringert. Da *cwnd* zu diesem Zeitpunkt 42 ist, wird *ssthres* auf 21 gesetzt (halber cwnd-Wert).
Hinweis: In Runde 17/18 ändert sich daran nichts. Vielleicht wundert man sich, warum cwnd nicht auch auf 21 gesetzt wird, sondern gleich auf 24? Der Grund: Der Paketverlust wurde durch 3 Duplicate ACKs erkannt (also viermal das gleiche ACK). Man geht aus, dass nur 1 Paket verlorengegangen ist und nicht 4. Deshalb 21 + 3!

p) In Runde 22 wird ein Problem erkannt als das *cwnd*=28. Deshalb wird ssthres in der 24. Runde den Wert 14 haben. Hinweis: Nur beim ersten Slow Start einer TCP Verbindung ist ssthres fest durch die Implementierung vorgegeben.

q) Paket 1 wird in der 1. Runde gesendet, Paket 2 und 3 in der 2. Runde, Pakete 4-7 in der 3. Runde, Pakete 8-15 in der 4. Runde, Pakete 16-31 in der 5. Runde. Das Paket 17 wird also in der 5. Runde gesendet.

Aufgabe 3: SSH Tunnel (Hausaufgabe, Demo in Übung)

a)
b)
c) IP Adressen, siehe Diagramm



d) Fügen Sie Default-Routen hinzu:
- lanHost: `ip route add default via 30.30.30.3 dev eth0`
- inetHost: `ip route add default via 40.40.40.4 dev eth0`
- relayHost: `ip route add default via 40.40.40.4 dev eth0`

e) Folgendes Kommando auf `lanHost` funktioniert: `ssh sshuser@40.40.40.3`
Folgendes Kommando auf `inetHost` funktioniert **nicht**: `ssh sshuser@30.30.30.2.` Der Grund hierfür ist, dass intern mit Firewall-Regeln Pakete von rechts nach links blockiert werden, sofern nicht zuvor eine Verbindung aufgebaut worden ist.

f) Den Reverse-Tunnel baut man mit folgendem Kommando auf:
lanHost: `ssh -N sshuser@40.40.40.2 -R 50000:localhost:22`
Ab sofort werden bei 40.40.40.2 ankommende TCP Verbindungen direkt auf Port 22 des lokalen Hosts also des lanHosts über den Tunnel weitergeleitet.
Der lanHost ist von außen / Internet deshalb mit folgendem Kommando per SSH erreichbar:
inetHost: `ssh -p 50000 sshuser@40.40.40.2`
Durch den Reverse Tunnel wurde sozusagen in Loch in das NAT-Gateway bzw. die Firewall gebohrt.

g) **lanHost:** `ssh -N sshuser@40.40.40.2 -R 50000:localhost:443`
inetHost: `ssh -p 50000 sshuser@40.40.40.2`