Semesterarbeit zur Vorlesung RN

Lukas Pietzschmann - 76010

31. Juli 2020

Abbildungsverzeichnis

1 Beispiel Webseite . . . 2
2 Aufgebaute TCP Verbindungen . . . 3
3 UDP Paket . . . 7
4 TCP Pakete . . . 8
5 TCP Handshake . . . 8
6 Congestion Control . . . 10

Inhaltsverzeichnis

1 Einleitung **2**

2 HTTP **2**
2.1 Persistent und non persistent HTTP . . . 2
2.2 HTTP Anfrage . . . 3
2.3 HTTP Antwort . . . 5

3 Die Transportschicht **6**
3.1 UDP . . . 6
3.1.1 UDP Segment Header . . . 6
3.2 TCP . . . 7
3.2.1 TCP Segment Header . . . 7
3.2.2 Handshake . . . 8
3.2.3 Neusenden von Paketen . . . 9
3.2.4 Flow-Control . . . 9
3.2.5 Congestion-Control . . . 10

Corona-Semester Vor- und Nachteile

Ich habe es dieses Semester sehr zu schätzen gewusst, dass ich mich morgens ganz gemütlich in Jogginghose und mit einer Tasse Kaffee an meinen Schreibtisch setzten konnte und nicht, wie sonst immer, in die Hochschule laufen musste.
Ein großer Nachteil war für mich hingegen, dass ich zu Hause viel leichter diversen Ablenkungen verfallen bin und es mir generell schwerer fiel mich länger zu konzentrieren.

Rechnernetze Vorwissen

Für diese Vorlesung hatte ich äußerst wenig Vorwissen. Sicher, kannte ich das OSI-Schichtenmodell und wusste grob, was TCP und UDP ist, aber mehr als simples Halbwissen hatte ich zu Beginn des Semesters nicht.

1 Einleitung

Diese Hausarbeit setzt sich mit drei Protokollen auseinander. So wird es um HTTP und dessen Requests und Responses gehen. Außerdem werden UDP und TCP beleuchtet bzw. v.a die verschiedenen Features von TCP.

2 HTTP

HTTP lässt sich folgendermaßen definieren:
HTTP (**H**yper**t**ext **T**ransfer **P**rotocol) ist ein zustandloses, textbasiertes Protokoll zur Datenübertragung auf der Applikationsschicht. Hauptsächlich wird es, im Rahmen des Client-Server Modells, von Browsern verwendet, um Webseiten in eben diese zu laden.

2.1 Persistent und non persistent HTTP

Derzeit sind hauptsächlich zwei Versionen des Protokolls in Verwendung. HTTP/1.0 und HTTP/1.1, veröffentlicht in 1996 bzw. 1999, unterscheiden sich, neben einigen anderen Features, vor Allem auch durch die Unterstützung von persistenten Verbindungen in HTTP/1.1. Dabei wird nicht für jede erneute `GET` Request eine neue TCP Verbindung aufgebaut. Somit verringert sich die Zeit für jede Anfrage ungleich der Ersten, um eine RTT. Möchte man also folgende Webseite laden,

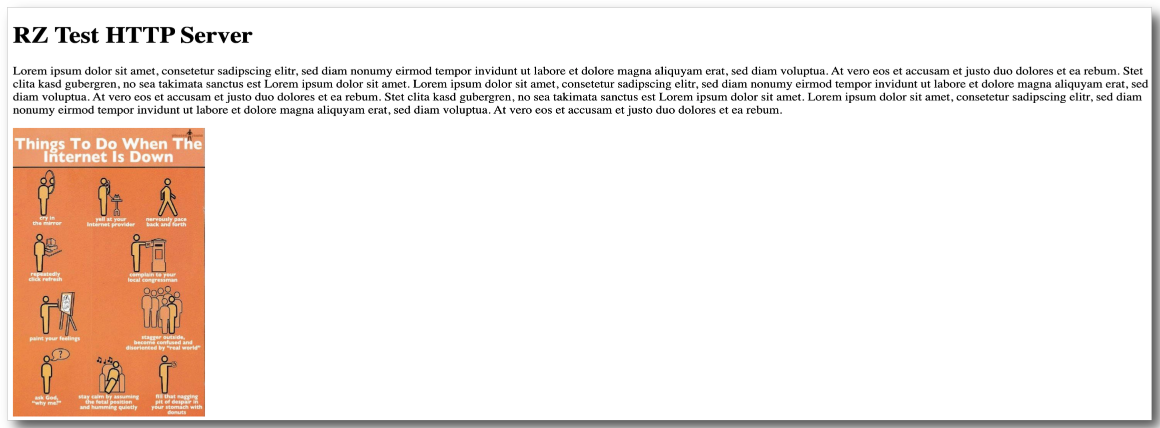

Abb. 1: Beispiel Webseite

muss separat der Text und das Bild angefragt werden und, da in diesem Beispiel HTTP/1.0 verwendet wird, zeigt folgendes Bild mit Wireshark wie zwei TCP Verbindungen aufgebaut werden.

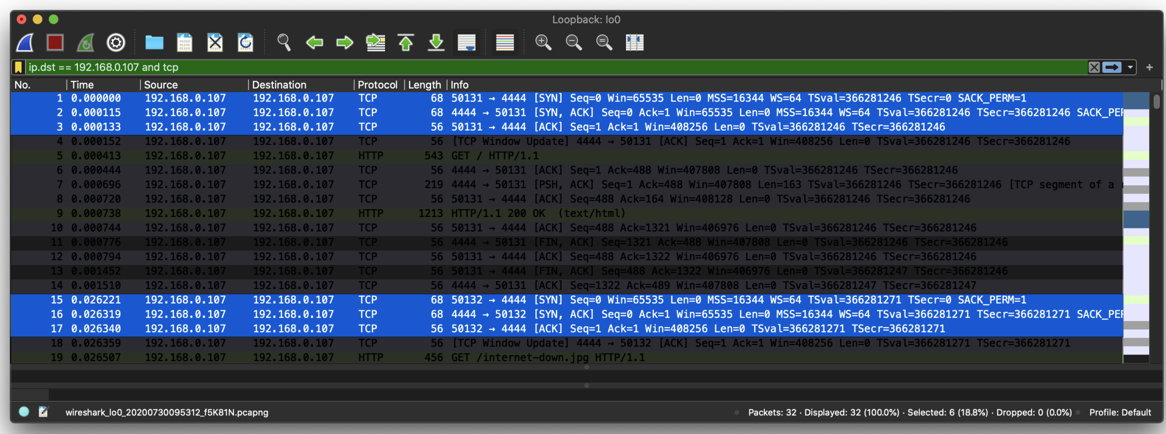

Abb. 2: Aufgebaute TCP Verbindungen

An den sechs blau markierten Zeilen, erkennt man anhand des TCP typischen drei Wege Handshake, dass pro HTTP Request eine Verbindung aufgebaut wird. HTTP/1.1 würde andererseits auf den zweiten Verbindungsaufbau verzichten.
Wenn dann eine TCP Verbindung besteht, wird als nächstes die erste HTTP Request versendet. Wie diese genauer aussieht, wird im Folgenden gezeigt.

2.2 HTTP Anfrage

Die GET Methode und diverse Header

Zur Veranschaulichung wird im Folgenden Burp als Proxy zum Abfangen der HTTP Anfrage und die selbe **Beispiel Webseite** wie im vorherigen Abschnitt verwendet.
Ruft man also in einem Browser diese Webseite auf, erzeugt der Browser folgende HTTP Nachricht.

Listing 1: HTTP Request

```
1  GET / HTTP/1.1
2  Host: 192.168.0.107:4444
3  Cache-Control: max-age=0
4  DNT: 1
5  Upgrade-Insecure-Requests: 1
6  User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_6) AppleWebKit
       /537.36 (KHTML, like Gecko) Chrome/84.0.4147.89 Safari/537.36
7  Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp
       ,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9
8  Accept-Encoding: gzip, deflate
9  Accept-Language: de-DE,de;q=0.9,en-US;q=0.8,en;q=0.7
10 Connection: close
```

Was an der Ausgabe von Burp leider nicht ersichtlich ist, ist dass ein Carriage-Return zusammen mit Line-Feed (\r\n) am Ende jeder Zeile steht bzw. stehen muss.
Die erste Zeile einer jeden HTTP Request ist die so genannte Request Line. Diese wird angeführt von dem konkreten Befehl (in diesem Fall GET), gefolgt von der URL zur gewünschten Datei. An dieser Stelle steht allerdings nie die komplette, vom Benutzer eingegebene URL, da ein Großteil dieser bereits in anderen Schichten aufgelöst wurde. Zuletzt muss noch die gewünschte HTTP

Version angegeben werden, Diese Versionsangabe ist jedoch unidirektional, was bedeutet, dass der Server nicht zwingend mit dieser Version antworten muss. So kann er auch in der nächst niedrigeren Version antworten, falls er die Gewünschte nicht unterstützt.
Danach folgen die Header der Anfrage. Diese bestehen aus, in einzelne Zeilen gefassten, Key-Value Paaren, beginnend mit der Spezifikation des Hosts. Die extra Angabe des Hosts wird vom Protokoll benötigt, falls es unter der adressierten IP mehrere Hosts gibt. Die darauf folgende Angabe des Cashing Verhaltens ermöglicht es dem Browser eine Ressource über einen angegebenen Zeitraum zwischenzuspeichern und wiederzuverwenden, anstatt eine erneute Anfrage für dieselbe Ressource zu stellen. Dies kann bei großen Dateien enorm Zeit einsparen. Zur genauen Spezifikation des Cashing Verhaltens stellt HTTP diverse Direktiven zur Verfügung. Die hier verwendete `max-age` Direktive gibt die Zeit der Zwischenspeicherung in Sekunden an. 0 verbietet hierbei das Cashing [2]. `DNT` in Zeile vier ist ein Akronym stehend für "Do Not Track". Wie der Name nahelegt wird hier mit einer 1 der Wunsch des Benutzers geäußert, lieber Privatsphäre, als personalisierten Content zu erhalten [1]. Auch `Upgrade-Insecure-Requests` dient dem Schutz der Privatsphäre. Hiermit gibt der Browser an, dass er das Upgrade auf eine sichere Verbindung unterstützt [4]. Der Server kann ihn nun zu einer sicheren Version der Seite weiterleiten. Eine solche Weiterleitung könnte folgendermaßen aussehen:

Listing 2: Weiterleitung zu HTTPS Seite

```
1  HTTP/1.1 200 OK
2  Location: https://192.168.0.107:4444
3  Vary: Upgrade-Insecure-Requests
```

Darauf folgend, gibt der Browser im `User-Agent` an, wer er ist. In diesem Beispielfall ist das der Google Chrome (Version 84). So kann der Server eine speziell für diesen Browser angepasste Version der angeforderten Webseite liefern. Die nächsten drei `Accept` Header spezifizieren die Sprache (`Accept-Language`), das erwünschte Encoding (`Accept-Encoding`) und welche Inhaltstypen präferiert werden. Wie bei `Accept-Language` gut zu sehen ist, lassen sich auch mehrere Präferenzen angeben, im Falle dass der Server die meist erwünschte nicht zur Verfügung stellen kann. Bei mehreren Angaben sollte jede einzelne mit Hilfe einer quality Value `q` gewichtet werden. Wichtig, es handelt sich dabei nur um eine Abstufung, nicht aber um eine prozentuale Angabe, die Summe über alle `q` muss also nicht 1 ergeben. Mit dem letzten Header wird dem Server mitgeteilt, dass man das, ab HTTP/1.1 verfügbare, Feature der persistenten Verbindung nicht nutzen möchte, sondern die Verbindung nach dieser Anfrage kappen möchte. `Connection: keep-alive` würde genau das Gegenteilige bezwecken.
Nach diesem langen Abschnitt über die vom Browser tatsächlich verwendeten Header, mag es enttäuschend sein zu lesen, dass all das optional ist. Die Zeile

Listing 3: Minimale HTTP Request

```
1  GET / HTTP/1.1
```

hätte, zumindest optisch, genau das selbe Ergebnis geliefert.

Andere HTTP Methoden

Das Beispiel aus **Listing 1: HTTP Request** enthielt in der Request Line die `GET` Methode. HTTP bietet allerdings noch diverse andere Methoden zur Interaktion zwischen Client und Server an.

HTTP/1.0

`HEAD` Bittet den Server, den Body der Antwort wegzulassen und lediglich die Header zu versenden. Eine mögliche Verwendung wäre beispielsweise eine Überprüfung auf

die Existenz einer Datei. Existiert diese dann tatsächlich, wird durch das nicht Senden der tatsächlichen Datei (sondern nur dem Header), Zeit eingespart

`POST` Hiermit kann eine Datei an den Server gesendet werden. Eine nähere Erläuterung findet sich später im Vergleich mit der HTTP/1.1 `PUT` Methode

HTTP/1.1

`GET`, `HEAD`, `POST` identisch zum HTTP/1.0 Äquivalent

`DELETE` Es wird die unter der URL angegebene Datei vom Server gelöscht. Laut Spezifikation, kann der Client aber selbst dann nicht sicher sein, dass der Vorgang tatsächlich ausgeführt wurde, wenn der empfangene Statuscode dies suggeriert. Dieses Verhalten kann auftreten, wenn `DELETE` serverseitig überschrieben worden ist [3].

`PUT` Diese Methode dient dazu eine Ressource auf einen Ziel Server hochzuladen. Eine nähere Erläuterung findet sich später im Vergleich mit der HTTP/1.1 `PUT` Methode

Unterschied PUT und POST
Wie bereits beschrieben, dienen sowohl `PUT` als auch `POST` dazu eine Ressource unter Angabe einer Ziel-URL auf einen Webserver hochzuladen. Der große Unterschied zwischen beiden ist, dass Folgendes

Listing 4: PUT / POST

```
1 [POST, PUT] /subfolder/<new_res> HTTP/1.1
2 Host:192.168.0.107:4444
```

bei `POST` einen Fehler wirft, da `POST` nur mit bereits bestehenden URL's arbeiten kann; Sprich, es können mit `POST` keine neuen Dateien erzeugt werden, mit `PUT` hingegen schon. Ein weiterer Unterschied ist, dass `POST` spezifizieren lässt wie die hochgeladene Datei (mit Hilfe eines Skripts) verarbeitet werden soll.

2.3 HTTP Antwort

Nun aber wieder zurück zu der ursprünglichen Anfrage an die Beispiel Webseite. Im folgenden wird, wieder mit Hilfe von Burp, die Antwort auf die **Request** betrachtet.

Listing 5: HTTP Response

```
1  HTTP/1.1 200 OK
2  Date: Wed, 29 Jul 2020 14:49:38 GMT
3  Connection: close
4  Content-Type: text/html; charset=UTF-8
5  Content-Length: 1157
6
7    <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/
        TR/html4/strict.dtd">
8    <html>
9    <head>
10   <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
11   <title>RZ Hausarbeit</title>
12   </head>
13   <body>
14   <h1>RZ Test HTTP Server</h1>
15   <p>
```

```
16   Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed    . . .
       . .
17   At vero eos et accusam et justo duo dolores et ea rebum.
18   </p>
19   <img src="internet-down.jpg" width="240" height="400">
20   </ul>
21   <hr>
22   </body>
23   </html>
```

Auch hier gilt wieder, dass jede Zeile mit einem Carriage-Return und einem Line-Feet terminiert wird. Beginnend also mit der sog. Status Line. Diese gibt an, dass wie erwünscht mit HTTP/1.1 geantwortet wurde, zusammen mit einem Statuscode. 200 ist dabei die Standartantwort, für eine erfolgreiche Anfrage. Gliedern lassen diese sich folgendermaßen:

1. 1xx: Informationen
2. 2xx: Erfolgreiche Anfrage
3. 3xx: Umleitung
4. 4xx: Fehler auf Client Seite
5. 5xx: Fehler auf Server Seite

In der zweiten Zeile wird die aktuelle Zeit, der Region des Servers angegeben, gefolgt von der Bestätigung, dass die TCP Verbindung nach der Antwort beendet wurde. Die Header `Content-Type` und `Content-Length` geben den Mime-Type und die Größe der folgenden Daten an. Oft wird dabei auch noch der Header `Accept-Range` zum spezifizieren der kleinsten Einheit der gesendeten Daten angegeben, ist dieser allerdings nicht angegeben ist davon auszugehen, dass es sich um Bytes handelt. Es folgt der Body der Antwort. Dass dieser so einfach lesbar ist, ist nicht zwingend gegeben. Der in diesem Beispiel enthaltene HTML Text, wird in nächsten Schritt dann direkt vom Browser gerendert und im Falle von referenzierten Elementen, werden diese mit einer erneuten HTTP Request angefragt.

3 Die Transportschicht

Zu den Aufgaben der Transportschicht zählt die Segmentierung des Datenstroms in einzelne Pakete und das Sicherstellen der Ankunft dieser. Die wohl meist bekannten Protokolle aus dieser Schicht sind UDP und TCP. Zuerst wird sich diese Arbeit mit UDP auseinandersetzen.

3.1 UDP

UDP lässt sich folgendermaßen definieren:
UDP (**U**ser **D**atagram **P**rotocol) ist ein verbindungsloses "best-effort" Protokoll. Verbindungslos bedeutet dabei, dass es keinen Handshake zwischen Sender und Empfänger gibt und jedes Paket unabhängig von allen anderen behandelt wird. "best-effort"beschreibt die Eigenschaft, dass nie garantiert ist, ob ein Pakte ankommt, oder dass Pakete in der korrekten Reihenfolge, oder auch nur einmal ankommen. Unter der Verwendung von UDP muss also die Anwendung selbst mit diesen fehlenden Garantien umgehen.

3.1.1 UDP Segment Header

Zum praktischen Zeigen des Headers, habe ich ein simples Client Server Skript in Python geschrieben, welches mittels UDP Nachrichten austauschen kann. Im Folgenden wird Wireshark verwendet um das UDP Paket näher zu betrachten. Vorweg ein kleiner Einschub zu Wireshark. Wichtig

ist hier zu wissen, dass Informationen die in eckigen Klammern dargestellt sind, von Wireshark berechnet worden sind, und so tatsächlich nicht im Segment stehen.

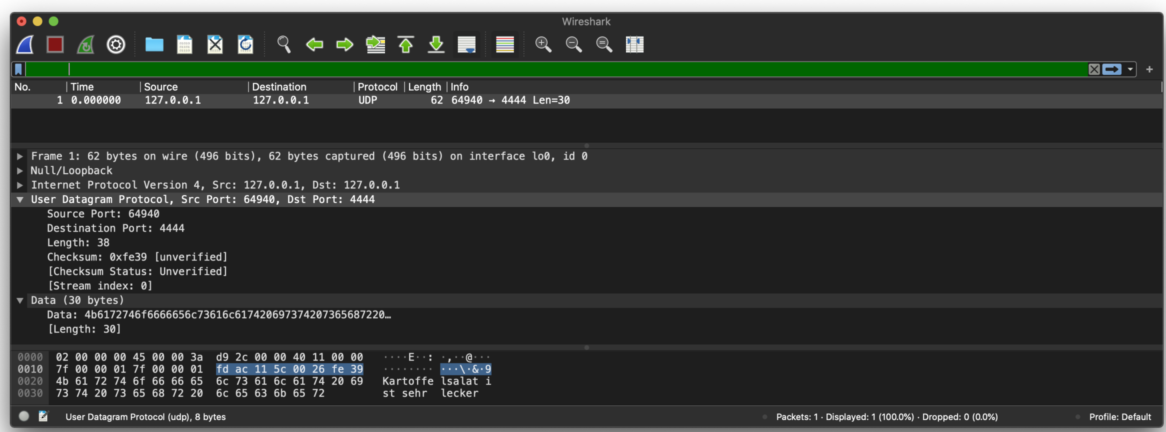

Abb. 3: UDP Paket

Damit lässt sich ersteinmal bestätigen, dass UDP verbindunglos ist. So (ganz oben im Bild zu sehen) wird nur ein einziges Paket gesendet, ergo gibt es keinen Handshake, sondern es wird direkt die Nachricht übermittelt. Als nächstes sieht man im mittleren Teil, den UDP Segment Header. Dieser ist 4 Byte breit und beginnt mit dem Source Port (2 Byte breit) . Dieser dient dazu ein Antwortpaket an den richtigen Port zu adressieren. Der darauf folgende Destination Port (2 Byte breit) gibt dem Betriebssystem des Ziels an an welche Applikation das Paket gerichtet ist. Es folgt die Länge des kompletten Headers (2 Byte breit), aus der sich dann auch die Länge der gesendeten Daten berechnen lassen (Länge - 8 Byte). Eine Checksumme (2 Byte breit) wird als nächstes benötigt, um mögliche Übertragungsfehler erkennen zu können. Für diese Checksumme wird das Segment als Folge von 16 Bit breiten Integern interpretiert und deren Einerkomplemente aufaddiert. Zuletzt werden um UDP Segment die reinen Bytes der gesendeten Daten angegeben.
Dadurch, dass UDP es ermöglicht ohne eine Verbindung aufzubauen direkt Daten über klein und simpel gehaltene Segmente zu schicken und somit keinen Overhead zu erzeugen, eignet sich UDP herrlich für einfache Frage-Antwort Anwendungen, wie beispielsweise DNS.
Im Folgenden wird als Kontrast TCP betrachtet, welches deutlich mehr Funktionalität und damit einhergehend auch deutlich mehr Komplexität bietet.

3.2 TCP

TCP lässt sich folgendermaßen definieren:
TCP (**T**ransmission **C**ontrol **P**rotocol) ist ein zuverlässiges, verbindungsorientiertes, bi-direktionales Protokoll. Es unterstützt sowohl Fluss-, als auch Verstopfungskontrolle.

3.2.1 TCP Segment Header

Zur Folgenden Untersuchung von TCP und dessen Segment Headers wurde mit netcat eine TCP Verbindung aufgebaut und eine einzige Nachricht gesendet.

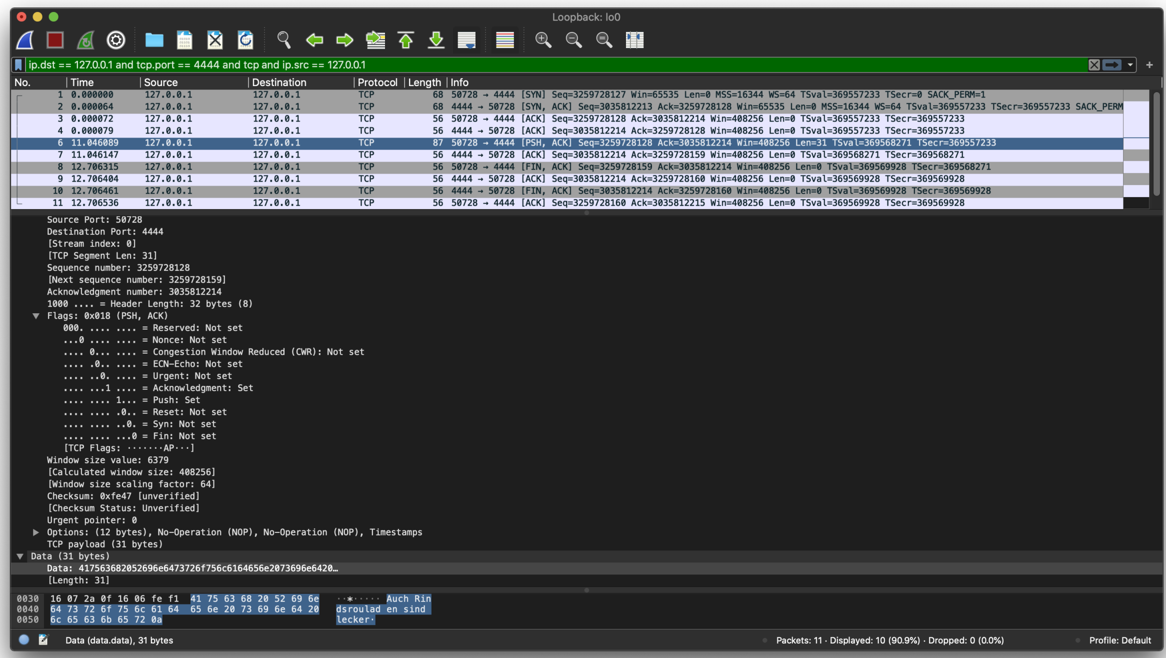

Abb. 4: TCP Pakete

Unter Betrachtung der ersten drei Pakete (Syn, SynAck, Ack) erkennt man den typischen TCP drei Wege Handshake, der notwendig ist, da TCP verbindungsorientiert ist. Nun aber zum Segment Header. Auch dieser ist 4 Byte breit. Wie auch bei UDP beginnt das Segment mit dem Ziel und Quell Port (2 Byte). Worauf direkt die Sequenznummer (4 Byte) folgt. Diese dient zur Sortierung der Pakete beim Empfänger in die richtige Reihenfolge, ist aber nur gültig falls das `SYN` Flag gesetzt ist. Die folgende Acknowledgementnummer (4 Byte) gibt an welches Paket man als nächstes erwartet. Auch diese ist aber nur gültig falls das `ACK` Flag gesetzt wurde. Schaut man in das Beispiel ist die Ack-Nummer des ausgewählten Pakets 3035812214 und die Seq-Nummer des nächsten Pakets dann auch tatsächlich 3035812214. Die Header Length (4 Bit) gibt, auch wie bei UDP, die Länge des kompletten Headers an. Als nächstes befinden sich die gesetzten Flags (1 Byte) im Header. Das sind in diesem Beispiel `PSH` und `ACK`. `PSH` steht dabei für Push und gibt an, dass in diesem Paket auch tatsächlich Daten gesendet wurden. Dies hilft dabei den TCP Datenstrom effizienter zu verarbeiten indem die Applikation beispielsweise nur bei gesetztem `PSH` Flag benachrichtigt wird. Das Recieve Window (2 Byte) Feld bestimmt die Anzahl an Bytes, die der Sender bereit ist zu empfangen. Es folgt die Checksumme (2 Byte), die identisch zu UDP berechnet wird. Allerdings werden bei TCP dazu nur die Felder Ziel-IP, Quell-IP, Protokollid (6) und die Länge des Headers verwendet. Der Urgent Data Pointer (2 Byte) zeigt auf Daten nach dem Header die mit höherer Priorität zu behandeln sind. Dieser Wert ist nur bei gesetztem `URG` Flag valide. Das letzte Headerfeld ermöglicht es zusätzliche Optionen festzulegen, die standardmäßig nicht im TCP Header vorhanden sind. Dieses Feld muss ein vielfaches von 32bit breit sein. Als letztes im Segment folgen nun die gesendeten Daten als rohe Bytes. Diese werden nur bei gesetztem `PSH` Flag auch tatsächlich verarbeitet.

3.2.2 Handshake

Wie bereits zu Beginn des vorherigen Abschnittes erwähnt implementiert TCP einen drei Wege Handshake.Im Folgenden werden die einzelnen drei Schritte Beschrieben.

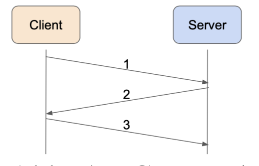

Abb. 5: TCP Handshake

1: SYN Hier wird die erste Sequenznummer `sc` bestimmt (nicht zwingend 1) und dem Server mitgeteilt, dass man eine Verbindung aufbauen möchte.

2: SYN ACK Im nächsten Schritt hat der Server das erste SYN Paket erhalten und bestimmt nun seine erste Sequenznummer `ss` (ebenfalls nicht zwingend 1). Dann acknowledged er das SYN Paket mit der ACK Nummer $\texttt{sc} + 1$.

3: ACK Der Client weiß nun, dass der Server bereit ist und bestätigt wiederum seine eigene Bereitschaft mit einem ACK $\texttt{ss} + 1$. Empfängt der Server das dann, wissen beide von der Bereitschaft des jeweils anderen und der Datentransfer kann beginnen.

Um eine Verbindung wiederum zu beenden wird erst von einer Seite ein Paket mit dem `FIN` Flag gesendet, welches dann wiederum ein ACK zurückerhält. Im letzten Schritt sendet dann die Gegenseite ebenfalls noch ein `FIN` welches auch bestätigt wird. Nun ist die Verbindung beendet. Am Beispiel oben gut an den letzten vier Paketen zu sehen.

3.2.3 Neusenden von Paketen

Was passiert nun aber falls ein Paket nicht durch ACK bestätigt wird? Dazu implementieren beide Seiten einen Timer, der für das älteste nicht bestätigte Paket runterzählt und erst zurückgesetzt wird, falls ein ACK ankommt. Ist dieser Timer abgelaufen muss das Paket erneut gesendet werden. Es ist sinnvoll den Wert des Timers etwas höher als die RTT zu wählen, nur ist die RTT kein fixer generell gültiger Wert. Aus diesem Grund wird die RTT gemessen und der Mittelwert gebildet. Wichtig ist, dass man beim Messen der RTT keine retransmissions misst, da zum Zeitpunkt des erneuten Sendens noch nicht bekannt ist, ob notwendigerweise neu gesendet wurde oder nicht. Und im Falle eines premature Timeouts würde das ACK für die erneute Sendung viel schneller ankommen. Die geschätzte RTT lässt sich also folgendermaßen berechnen:

$$EstRTT = (1 - \alpha) \cdot EstRTT + \alpha \cdot SampleRTT$$

α ist dabei typischerweise ca. 0,125. Das Timeout bestimmt man also wie Folgt:

$$Timeout = EstRTT + 4 \cdot Margin$$

Je größer die Veränderungen von $EstRTT$ sind desto größer soll auch $Margin$ sein. Erhält man dann aber das ACK, wird der Timer nur neugestartet, falls es immer noch unbestätigte Pakete gibt. Ein erneutes Senden kann neben dem Timeout auch durch das doppelte erhalten des selben ACKs (mit der selben Ack-Nummer) zustande kommen. Falls das der Fall ist und der Timer aber noch nicht abgelaufen ist, ist ein duplicate ACC das stärkere Kriterium und des wird direkt erneut gesendet.

3.2.4 Flow-Control

Fluss Kontrolle bezieht sich auf die Empfänger Seite. So möchte der Empfänger dem Sender mitteilen, dass sein Buffer an unbestätigten Nachrichten voll ist, der Sender also nichts mehr bzw. weniger senden soll. Anstatt erst aktiv zu werden, wenn der Buffer bereits überläuft, verwendet der Empfänger hierbei den Recieve Window Header um anzugeben wie viel Platz in seinem Puffer noch frei ist. Der Sender kann so die Menge an Paketen die er schickt so weit reduzieren, dass die angegebene Größe nicht überschritten wird. Im Beispiel von oben, ist die Recieve Window Size durchgehend 6379 Byte. Dem Buffer droht also zu keinem Zeitpunkt ein Überlauf.

3.2.5 Congestion-Control

Verstopfungskontrolle bezieht sich im Gegensatz zur Fluss Kontrolle auf den Sender. Zur Implementierung von Congestion-Control verwendet TCP ein Congestion-Window (cwd), das spezifiziert, wie viele unbestätigte Pakete gesendet werden dürfen. So beginnt TCP in der sog. Slow Start Phase. Hier entspricht das cwd genau der Länge eines maximal langen Pakets (MSS), wird dann aber nach jeder RTT so lange verdoppelt, bis der erste Verlust registriert wird. Wird der Verlust durch einen Timeout registriert, reagieren alle TCP Implementationen gleich: Das cwd wird wieder auf 1 MSS gesetzt, bis zu einem bestimmten Schwellwert exponentiell verdoppelt (danach nur noch linear) und bei einem Verlust, wird dann wieder wie in diesem Abschnitt beschrieben reagiert. Der eben genannte Schwellwert wird im Falle eines Verlusts auf $\frac{cwd}{2}$ gesetzt, damit beim nächsten mal ab dem Zeitpunkt des letzten Verlusts nur noch linear erhöht wird. Wird der Verlust allerdings durch ein dreifaches doppeltes ACK erkannt können verschiedene Implementierungen auch unterschiedlich reagieren. Dazu werden im Folgenden TCP Reno und TCP Tahoe betrachtet. TCP Reno interpretiert doppelte ACKs so, dass das Netzwerk ja fähig ist, eine gewisse geringe Anzahl an Paketen zu schicken (sonst wären die doppelten ACKs nicht angekommen). Also halbiert Reno das cwd nur und erhöht danach wieder linear. Tahoe hingegen verfährt wie im Falle eines Timeouts.

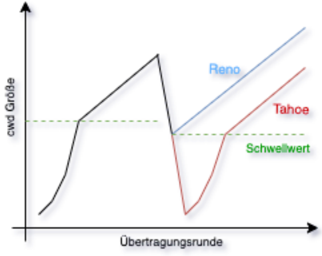

Abb. 6: Congestion Control

Zusammenfassung und wertvollste Erkenntnisse

Zusammenfassend kann man, bei HTTP beginnend, also sagen, dass wir dort gelernt haben was der Unterschied zwischen persistentem HTTP und nicht persistentem HTTP ist. Wir haben verschiedene Methoden und Header kennengelernt und wie man diese in einer HTTP Request zusammenfasst. Auch haben wir und mit der HTTP Response und deren Statuscodes beschäftigt. Meine wertvollste Erkenntnis in diesem Thema war es, diese vermeintliche "Magie"des Browsers aufzulösen und zu sehen was er tatsächlich sendet. Bei UDP haben wir uns angeschaut wie der UDP Segemt Header denn aussieht und an einem praktischen Beispiel wie dessen Checksumme berechnet wird. Bei TCP haben wir auch mit dessen Segmentstruktur begonnen um dann mit der konkreten Umsetzung von Ackknowladgements weiter zu machen. Auch haben wir uns damit beschäftigt wie eine zuverlässige Verbindung gewährleistet werden kann und wie gehandelt wird, falls es auf Sender oder Empfängerseite zu Verstopfungen kommt. Meine wertvollste Erkenntnis bei TCP und UDP war es, wie genau Daten aufgeteilt werden und dann tatsächlich durch die Internetleitung geschickt werden.

Literatur

[1] Peter Eckersley. “What Does the "Track̈ın "Do Not Track"Mean?” In: (2011). URL: https://www.eff.org/deeplinks/2011/02/what-does-track-do-not-track-mean.

[2] Ilya Grigorik. “HTTP-Caching”. In: (2018). URL: https://developers.google.com/web/fundamentals/performance/optimizing-content-efficiency/http-caching?hl=de.

[3] Network Working Group. “Method Definitions”. In: (1999). URL: https://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html.

[4] Mike West. “Upgrade Insecure Requests”. In: (2015). URL: https://www.w3.org/TR/upgrade-insecure-requests/.