Belegarbeit Internettechnologien 2

Vorname, Name: Leonard Hecker
Anschrift: August-Bebel-Str. 44, 01219 Dresden

14. Januar 2018

1 Prolog

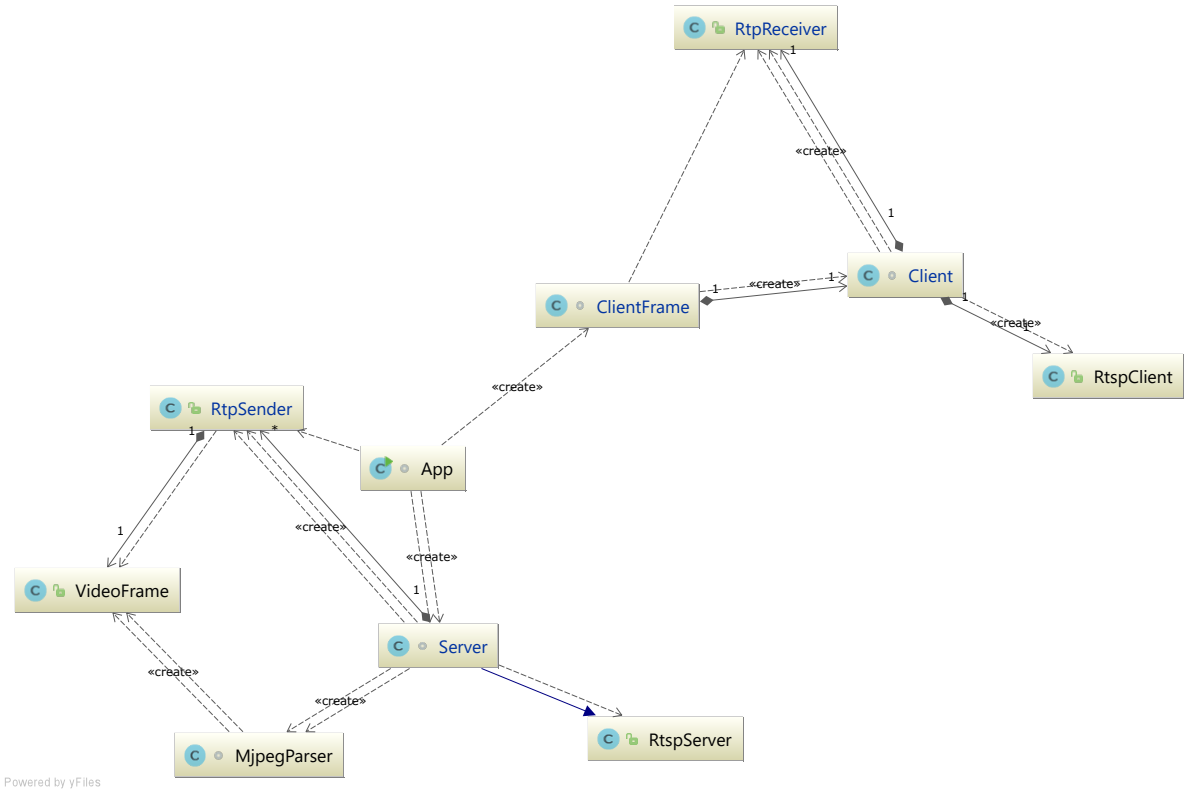
Das vorliegende Projekt stellt eine nahezu vollständig konforme Implementierung der für die Aufgabenstellung wesentlichen Bestandteile von RTSP [1], RTP [2], sowie RTP-basierter FEC [3] dar.

Das Projekt wurde innerhalb der Domain `hecker.io` in 3 Packages unterteilt:

- `rtsp`: Eine generische RTSP Client/Server Implementierung
- `rtp`: Eine partiell generische RTP Sender/Receiver Implementierung, samt Unterstützung für MJPEG und FEC Pakete
- `it2`: Kombiniert die 2 überliegenden Packages um letztendlich die geforderten Fähigkeiten in Form einer CLI/GUI Anwendung zu implementieren

Es sei des Weiteren erwähnt, dass viele der enthaltenen Klassen von guava's `AbstractExecutionThreadService` erben. Solcherlei Klassen erfüllen guava's `Service` Interface zum asynchronen Start, Stop sowie zur Überwachung des Servicezustands. `AbstractExecutionThreadService` im besonderen führt hierbei zur Erfüllung des Interfaces die abstrakte Methode `run` in einem seperaten Worker-Thread aus.

2 io.hecker.it2



An zentraler Stelle steht die Klasse `App`, welche die `main`-Funktion enthält. Zunächst überträgt diese die `--loss` und `--fec` Parameterwerte an die `RtpSender` Klasse, damit diese später einen entsprechenden Paketverlust simuliert bzw. FEC-Korrektur anbietet. Des Weiteren wird abhängig von dem `--server` Flag entweder eine `Server`, oder eine `ClientFrame` Instanz erzeugt, welche alle weiteren Tätigkeiten vornimmt.

2.1 Server

Die `Server` Klasse erbt von `RtspServer` und implementiert mithilfe dieser die geforderten, konkreten Methoden `OPTIONS`, `DESCRIBE`, `SETUP`, `TEARDOWN`, `PLAY` und `PAUSE`.

Wird ein `SETUP` Request erhalten, so wird mithilfe der Klasse `MjpegParser` die im Request-Pfad angegebene Datei - in diesem Fall `sample.mjpeg` - geöffnet und später in einzelne `VideoFrame`s geparsed. Zeitgleich wird eine neue `RtpSender` Instanz erzeugt, welche später eben jene `VideoFrame`s konsumiert, jeweils in `RtpPacket`s umwandelt und diese als UDP Pakete zum angegebenen Empfänger sendet. Bis zum Empfang eines `PLAY` Requests ist jedoch der `RtpSender` zunächst in einem pausierten Zustand. Der `Client` erhält nun vom `Server` eine Session ID, welche für weitere Requests verwendet werden kann.

Erhält der `Server` einen Request, so wird dieser mithilfe der übergebenen Session ID den entsprechenden `RtspSender` suchen und diesen zerstören.

`TEARDOWN`, `PLAY` und `PAUSE` Requests hingegen sind relativ simpel implementiert und suchen jeweils nur den zur, im Request angegebenen, Session ID entsprechenden `RtspSender` und zerstören, de-pausieren bzw. pausieren ihn.

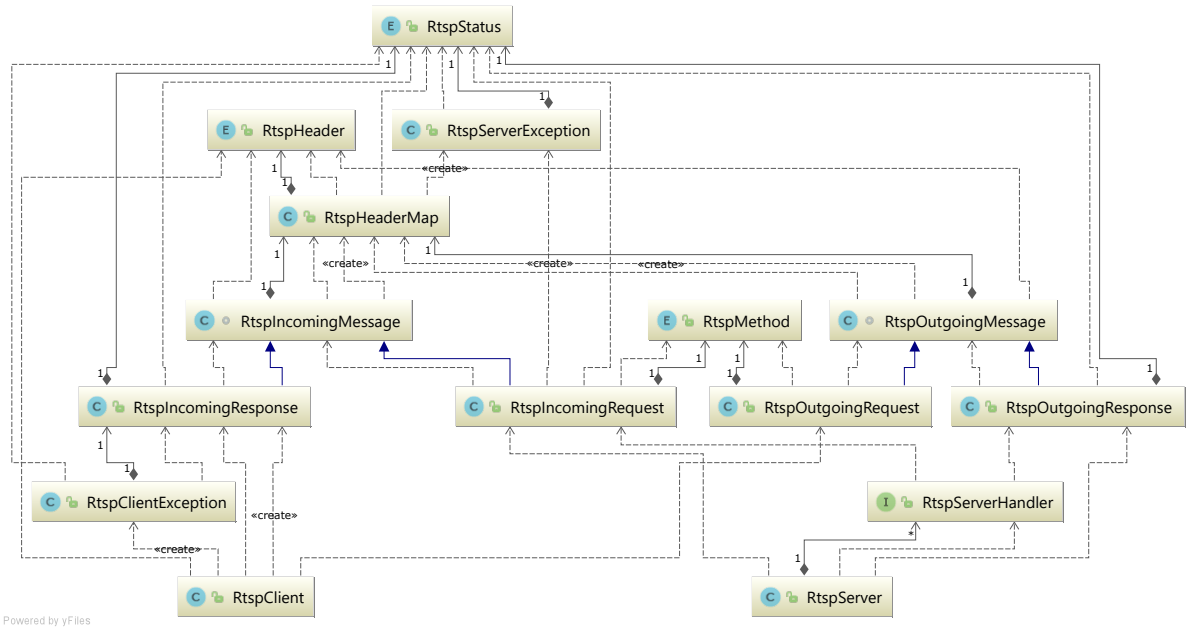
2.2 ClientFrame

Die `ClientFrame` Klasse erbt von `JFrame` um eine GUI Anwendung zu erzeugen. Sie nutzt primär die namensgebende `Client` Klasse um jegliches RTSP, sowie RTP Handling zu vollführen und leitet alle UI-Interaktionen stellvertretend an diese weiter.

2.3 Client

Die `Client` Klasse erzeugt während des Startups einen `RtpReceiver` und sendet einen `SETUP` Request an den angegeben Server. Analog wird während des Shutdowns ein `TEARDOWN` Request durchgeführt. Die hierdurch beim Setup vom Server erhaltene Session ID wird nun für alle weiteren Abläufe verwendet. Klickt der Nutzer auf das Video-Canvas im `ClientFrame`, wird, abhängig davon ob das Video aktuell pausiert ist oder nicht, ein entsprechender `PLAY` bzw. `PAUSE` Request gesendet.

3 io.hecker.rtsp



In der RTSP [1] Implementierung stehen an zentraler Stelle die Klassen `RtspClient` sowie `RtspServer`.

3.1 RtspClient

`RtspClient` implementiert einen generischen RTSP Client. Zur Durchführung eines Request mittels der `fetch` Methode wird eine Instanz der Klasse `RtspOutgoingRequest` übergeben. Nach der internen, automatischen Generierung des `CSEQ` Headers wird dann der Request serialisiert und gesendet. Daraufhin wird eine `RtspIncomingResponse` erzeugt und mithilfe dieser die Response des Servers eingelesen, deserialisiert und zurückgegeben.

3.2 RtspServer

`RtspServer` implementiert einen generischen RTSP Server. Bei diesem können mittels der `addHandler` Methode eine beliebige Anzahl an `RtspServerHandler` Callbacks registriert werden, welche dann für jeden Request aufgerufen werden, diesen inspizieren und Response-Daten setzen können.

Wurde eine TCP Connection angenommen, wird nun für jede einzelne ein `RtspServer.Connection` Service gespawnt. Dieser erzeugt daraufhin in einer Dauerschleife `RtspIncomingRequest` Instanzen und liest und deserialisiert mit diesen einzelne Requests. Zusammen mit einer noch leeren `RtspOutgoingResponse` Instanz wird so bei jedem Request nun die Liste aller registrierten `RtspServerHandler` aufgerufen. Die vorliegende Server-Implementierung fügt hierbei automatisch bereits einige Standard-Header

hinzu, wie z.B. den strikt verlangten `Date` Header, sowie `CSEQ` und `Content-Type`. Die `RtspOutgoingResponse` Instanz wird letztendlich serialisiert und an den Client gesendet.

3.3 RtspServerHandler

Das `RtspServerHandler` Interface wird vom Server genutzt um bei diesem Callbacks bzw. Handler zu registrieren. Jeder `RtspServerHandler` wird dann vom Server mit den aktuellen Request/Response-Paar aufgerufen und erlaubt dem Anwender den Request zu inspizieren und die Response beliebig mit Daten zu füllen.

Es ist hierbei möglich eine `RtspServerException` zu werfen. Diese wird vom `RtspServer` aufgefangen und der darin enthaltene Status Code in der Response gesetzt.

3.4 RtspIncoming*

`RtspIncomingRequest` sowie `RtspIncomingResponse` erben von `RtspIncomingMessage`, welches die Deserialisierung der Headerzeilen mittels einer `RtspHeaderMap` sowie die des Bodys implementiert. `RtspIncomingRequest` und `RtspIncomingResponse` fügen hierbei nur noch das Einlesen der jeweiligen Statuszeile hinzu um die Funktionalität zu vervollständigen.

3.5 RtspOutgoing*

`RtspOutgoingRequest` sowie `RtspOutgoingResponse` erben von `RtspOutgoingMessage`, welches die Serialisierung der Headerzeilen mittels der enthaltenen `RtspHeaderMap` sowie die des Bodys implementiert. `RtspOutgoingRequest` und `RtspOutgoingResponse` fügen hierbei nur noch das Erzeugen der jeweiligen Statuszeile hinzu um die Funktionalität zu vervollständigen.

3.6 RtspHeaderMap

`RtspHeaderMap` stellt ein vereinfachtes `Map` Interface für Request/Response Header dar. Des Weiteren wird hier die Deserialisierung sowie Serialisierung der Headerzeilen von einem `DataInput` sowie in ein `DataOutput` implementiert.

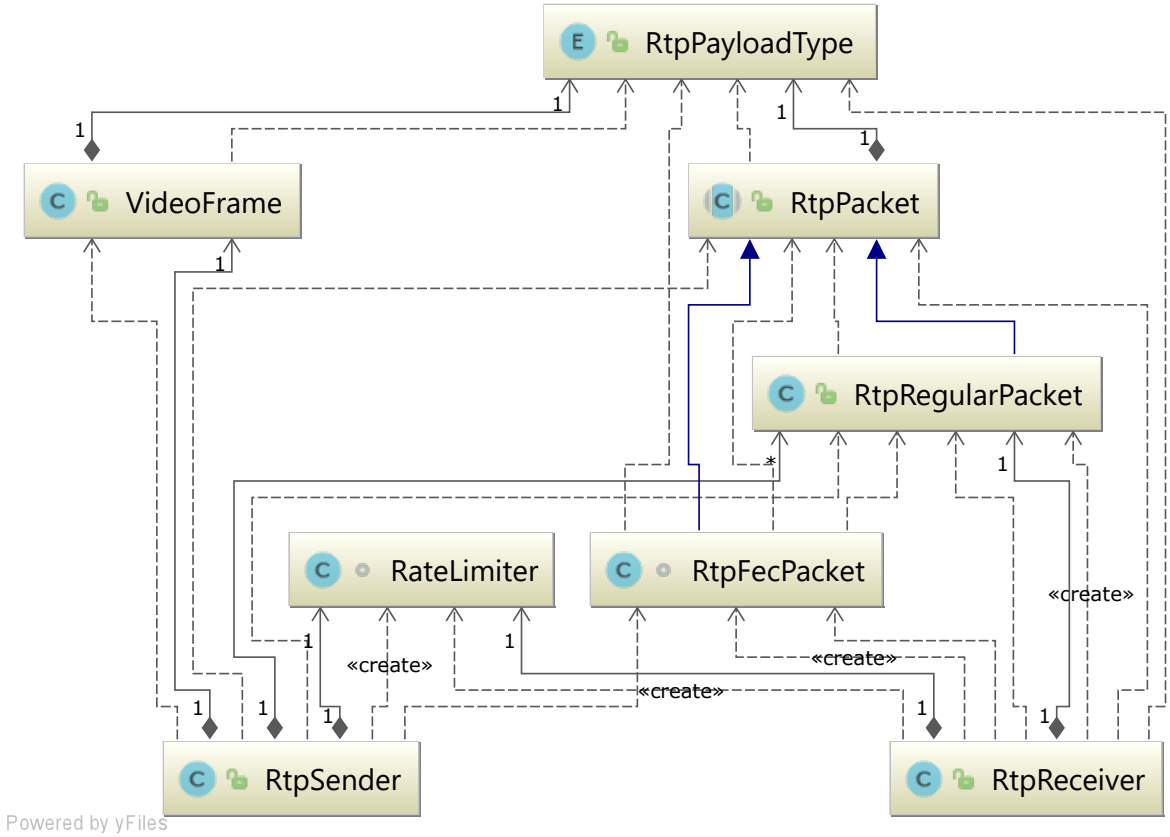
3.7 RtspStatus

Das Enum `RtspStatus` enthält alle unterstützten RTSP Statuscodes.

3.8 RtspMethod

Das Enum `RtspMethod` enthält alle unterstützten RTSP Anfragemethoden.

4 io.hecker.rtp



In der RTP [2] Implementierung stehen an zentraler Stelle die Klassen `RtpSender` sowie `RtpReceiver`.

4.1 RtpSender

Die `RtpSender` Klasse wird mit einer Zieladresse und einem `VideoFrame` Iterator konstruiert, wandelt letzteres in eine Serie von `RtpPacket` Instanzen um und sendet diese an die angegebene Zieladresse.

Die Implementierung des `RtpSender` ist weitestgehend sehr simpel. Mittels einem `RateLimiter` wird die Rate des Einlesens vom `VideoFrame` Iterator und des Sendens derart gesteuert, dass dies der Framerate des Videos entspricht. Wurde ein `RtpRegularPacket` gesendet, so wird dieses nur noch optional der FEC Queue hinzugefügt.

Sofern das `--fec` Flag gesetzt und der äquivalente `FEC_SIZE` Wert größer 0 ist, wird nun das Paket der `m_fecQueue` hinzugefügt. Enthält die `m_fecQueue` mindestens `FEC_SIZE` Elemente, so werden die darin enthaltenen Pakete zu einem einzelnen `RtpFecPacket` kodiert und dem Empfänger gesendet.

4.2 RtpReceiver

Die `RtpReceiver` Klasse ist ein `Service`, welcher asynchron im Hintergrund UDP Pakete empfängt und diese mittels der `RtpPacket` Klasse etc. deserialisiert.

Primär empfängt ein `RtpReceiver` Pakete des Typs `RtpRegularPacket`. Diese werden hierbei in der Queue `m_queue` für eine spätere entnahme abgelegt. Die Queue ist hierbei eine `PriorityQueue` welche anhand der Sequence Number bzw. alternativ anhand des Timestamp der enthaltenen Pakete sortiert wird. Auf diese Weise werden später Out-of-Order Pakete trotz dessen in der richtigen Reihenfolge entnommen.

Wird ein `RtpFecPacket` empfangen, wird diesem die aktuelle Queue übergeben, mithilfe welcher die `RtpFecPacket` Implementierung bis zu ein verloren gegangenes Paket widerherstellen kann.

Im gegenüberliegenden Fordergrund steht die Methode `next` mit welcher Pakete aus der Queue entnommen werden können. Hierbei steht der Monitor `m_queueMonitor` an zentraler Stelle zur Synchronisation mit dem im Hintergrund laufenden Service.

Enthält die Queue weniger als 16 Pakete, z.B. weil der Stream noch nicht gestartet hat, so wird zunächst gewartet bis dies der Fall ist. Die hierbei verwendete Logik erlaubt es zusätzlich bis zu 3 s zu warten bis die Queue optional bis zu 32 Pakete enthält. Dies stellt einerseits sicher, dass immer alle bis zu 16 Pakete, welche von einem `RtpFecPacket` referenziert werden können, in der Queue auffindbar sind und andererseits, dass beim Start des Streams ein möglichst großer Buffer vorhanden ist um etwaige Latenzschwankungen etc. auszugleichen.

Enthält die Queue nun genügend Pakete, so wird äquivalent zum `RtpSender` mittels einem `RateLimiter` und dem Monitor gewartet bis das nächste RTP Paket angezeigt werden kann. Ändert sich aufgrund von Out-of-Order Paketen oder FEC Reperatur die Spitze der Queue und somit das voraussichtlich als nächstes anzuzeigende Paket, so ändert sich demzufolge auch die Wartezeit. Dies wird mittels dem `QueueHeadChangedGuard` und dem Monitor überwacht und in einem solchen Falle der Warteprozess mit dem Timestamp des neuen Queue Head von neuem gestartet.

4.3 RtpPacket

Die abstrakte Klasse `RtpPacket` implementiert die gemeinsame Funktionalität der Klassen `RtpRegularPacket` und `RtpFecPacket`: Die Serialisierung, Deserialisierung und einen Builder für alle statischen Felder eines RTP Packets, definiert in RFC3550 [2], Sektion 5.1. Der optionale Extension Header, sowie das Auslesen von `CSRC` Feldern wurde hierbei jedoch nicht implementiert.

Diese Klasse und ihre beiden Implementierungen machen sich des Weiteren das Builder-Pattern zu nutze, um elegant immutable Instanzen zu erzeugen.

4.4 RtpRegularPacket

`RtpRegularPacket` ist eine schlanke Erweiterung des `RtpPacket` und fügt einzig das Serialisieren und Deserialisieren des Payloads hinzu.

4.5 RtpFecPacket

`RtpFecPacket` implementiert RTP-basierte FEC. Während die Implementierung der Serialisierung und Deserialisierung von Paketen vollständig RFC5109 [3], Sektion 7, folgt, bietet die triviale Implementierung der Recovery jedoch nur Unterstützung von FEC Level 0.

4.6 RateLimiter

Die Klasse `RateLimiter` dient dem `RtpSender` sowie dem `RtpReceiver` dazu, die Rate des Sendens bzw. Anzeigens zu steuern. Die Klasse implementiert im Kern den folgenden Algorithmus:

$$P_{n+1} = P_n + (T_{n+1} - T_n)$$
$$S = P_{n+1} - Z_n$$

Mit:

- P: Zeitstempel der letzten Präsentation
- T: Paketzeitstempel
- Z: Aktuelle Zeit
- S: Dauer die geschlafen werden muss um das Rate Limit zu erfüllen

Dies stellt sicher, dass kleinere temporale Fluktuenzen zwischen den Aufrufen der Methoden des `RateLimiter` herausgefiltert werden und somit, dass über die Dauer des Streamings hinweg kein Zeitdrift auftritt.

5 Epilog

Ein zufriedenstellender Wert für k wurde auf theoretische Weise ermittelt. Die `sample.mjpeg` Datei ist 4 269 893 B groß und enthält 500 Frames mit jeweils einem 5 B Header. Dies entspricht einer durchschnittlichen Payloadgröße von 8535 B, sowie der kumulativen Gesamtgröße aller Frames von 4 267 393 B. Aufgrund der signifikanten Größe des durchschnittlichen Payloads wird im folgenden die Größe des RTP und FEC Headers in Berechnungen vernachlässigt.

Des Weiteren werden die Konstanten

$$p = 0.1$$
$$G = 4\,267\,393\,\mathrm{B}$$

angenommen.

k	Qualität	Datenmenge	Q_q	Q_d	Score
2	0.99	5760981	1.00	0.00	
3	0.97	5120872	0.96	0.38	0.37
4	0.95	4800817	0.91	0.57	0.52
5	0.92	4608784	0.85	0.69	0.58
6	0.89	4480763	0.78	0.76	0.59
7	0.85	4389319	0.71	0.82	0.58
8	0.81	4320735	0.63	0.86	0.54
9	0.77	4267393	0.55	0.89	0.49
10	0.74	4224719	0.47	0.91	0.43
11	0.70	4189804	0.38	0.94	0.36
12	0.66	4160708	0.30	0.95	0.29
13	0.62	4136089	0.22	0.97	0.22
14	0.58	4114986	0.15	0.98	0.14
15	0.55	4096697	0.07	0.99	0.07
16	0.51	4080695	0.00	1.00	

Die *Qualität* entspricht hierbei der Wahrscheinlichkeit, dass ein Paket verloren geht. Dies wurde mithilfe von Bernoulli's unteren, kumulativen Verteilungsfunktion $F(k, n, p) = P(X \leq k)$ berechnet. Hierbei entsprechen, gemäß der üblichen Benennung der Parameter der Verteilungsfunktion, der Parameter k der Anzahl an Erfolgen, welcher im Falle einer gerade noch gelingenden Level 0 FEC Recovery 1 beträgt, der Parameter n den in der Aufgabenstellung und in der obigen Tabelle verwendeten Parameter k, sowie p der konstanten Verlustrate 0.1.

Die *Datenmenge* D berechnet sich des Weiteren mittels:

$$D = \left(G + \frac{G}{k}\right) * (1 - p)$$

Hierbei wurde die Berechnung des Overheads der FEC Pakete vereinfacht und mit dem Näherungswert $G \div k$ ausgedrückt.

Die Spalten $Q_{q/d}$ sollen hierbei auf einfache Weise die Qualitätsmerkmale quantifizieren. Hierbei wird Q_q, dem Quantum der Qualität, sowie Q_d, dem Quantum der Datenmenge,

die Werte 1 für das Beste und 0 für das Schlechteste jeweilige Ergebnis zugeordnet. Alle zwischenliegende Werte erhielten ein linear interpolierten Wert.

Der endgültige *Score* entspricht dem Produkt $Q_q * Q_d$.

Hierbei stellt sich $k = 6$ *als ein gute Balance zwischen Datenmenge und Qualität heraus.*

Literatur

[1] H. Schulzrinne, A. Rao und R. Lanphier. *Real Time Streaming Protocol (RTSP)*. RFC 2326. `https://www.rfc-editor.org/rfc/rfc2326.txt`. RFC Editor, Apr. 1998. URL: `https://www.rfc-editor.org/rfc/rfc2326.txt`.

[2] H. Schulzrinne u. a. *RTP: A Transport Protocol for Real-Time Applications*. STD 64. `https://www.rfc-editor.org/rfc/rfc3550.txt`. RFC Editor, Juli 2003. URL: `https://www.rfc-editor.org/rfc/rfc3550.txt`.

[3] A. Li. *RTP Payload Format for Generic Forward Error Correction*. RFC 5109. `https://www.rfc-editor.org/rfc/rfc5109.txt`. RFC Editor, Dez. 2007. URL: `https://www.rfc-editor.org/rfc/rfc5109.txt`.