Camera Stream Solution
–
Marktübersicht, Lösungsansätze, Prototyp

Masterarbeit

von

Saskia Schlingensief

bei der Herrenknecht AG

zur Erlangung des akademischen Grades

Master of Science (M.Sc.)

im Studiengang Medien und Kommunikation

Fakultät Medien und Informationswesen

Hochschule Offenburg

eingereicht im Wintersemester 2022/23

am 30.01.2023

Bearbeitungszeitraum vom 01.09.2022 bis 01.03.2023

Erstgutachter: Prof. Dr. Volker Sänger (Hochschule Offenburg)

Zweitgutachter: Dipl.-Ing. Torsten Weiser (Digitale Produkte & Services, Herrenknecht AG)

HERRENKNECHT
Tunnelling Systems

Hochschule Offenburg
offenburg.university

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Masterarbeit zum Thema „Camera Stream Solution – Marktübersicht, Lösungsansätze, Prototyp“ selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel verfasst habe.

Alle Stellen der Arbeit, die wörtlich oder sinngemäß aus Veröffentlichungen oder aus anderweitigen fremden Äußerungen entnommen wurden, sind als solche kenntlich gemacht.

Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Gengenbach, 30.01.2023

_____

Saskia Schlingensief (Unterschrift)

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

ABR-Streaming ..... *Adaptive Bitrate-Streaming*
API ..... *Application Programming Interface*
fps ..... *frames per second*
GOP ..... *Group of Pictures*
HLS ..... *HTTP Live Streaming*
HOL ..... *Head of line blocking*
MPD ..... *Media Presentation Description*
QUIC ..... *Quick UDP Internet Connections Protocol*
RTCP ..... *Real-time Transmission Control Protocol, RTP Control Protocol*
RTMP ..... *Real-Time Messaging Protocol*
RTP ..... *Real-Time Transport Protocol*
RTSP ..... *Real-Time Streaming Protocol*
RTT ..... *Round Trip Time*
TBM ..... *Tunnelbohrmaschine*
TCP ..... Transport Control Protocol
UDP ..... User Datagram Protocol
VNC ..... *Virtual Network Computing*
VPN ..... *virtuelles privates Netzwerk*
W3C ..... *World Wide Web Consortium*
WebRTC ..... *Web-Real-Time-Communication*
WSL ..... *Windows-Subsystem für Linux*

1. Einleitung

> *„… the streaming media technology [is becoming] the most important transport technology of online multimedia information, especially video information.“ [1]*

In diesem Zitat von Wang wird beschrieben, wie wichtig das Streaming zur Übertragung von Multimediadaten im Internet in der Gegenwart ist. Diese Form der Datenübertragung gibt es zwar bereits seit den 1990er Jahren, allerdings hat sie insbesondere seit der Einführung von YouTube im Jahr 2005 große Bedeutung erhalten und wird für verschiedene Anwendungen wie zum Beispiel Video-On-Demand-Angebote, Videokonferenzen, Fernunterricht, Unterhaltungsangebote wie Gaming oder Internet-TV eingesetzt [2]. Daher gilt: „Video technology now powers everything, for everyone, everywhere.“ [3]

Um das Streaming beziehungsweise das Videostreaming geht es auch in dieser Thesis mit dem Thema „Camera Stream Solution – Marktübersicht, Lösungsansätze, Prototyp“. Das Projekt wurde in Zusammenarbeit mit der Herrenknecht AG bearbeitet, um die Kundenplattform Herrenknecht *CONNECTED* um eine Videostreaming-Lösung zu erweitern. Dabei geht es um die Übertragung in nahezu Echtzeit von Inhalten der Navigations- und Steuerungsbildschirme, die sich auf aktiven Tunnelbohrmaschinen befinden, sodass Kunden Einblick in diese haben, ohne dass sie sich auf den Maschinen befinden müssen.

Das Ziel in dieser Thesis ist es, das Videostreaming für mindestens einen Testrechner prototypisch umzusetzen und eine Möglichkeit zu finden, die Lösung großflächig für alle aktiven Maschinen anbieten zu können.

Eingesetzt werden verschiedene Streaming-Komponenten, um den Bildschirm auf der Tunnelbohrmaschine aufzunehmen, die Aufnahme als Stream an einen Server zu schicken und schließlich auf *CONNECTED* bereitzustellen, sodass Kunden ihn quasi in Echtzeit abrufen können.

Die Streaming-Lösung soll Kunden sowie internen Herrenknecht-Mitarbeitern einen Einblick in die Cockpit-Monitore der Maschinenfahrer geben. Dabei handelt es sich zum Beispiel um Navigationsbildschirme, um die aktuelle Position der Tunnelbohrmaschine anzusehen, oder weitere Ansichten, die Parameter wie die Anpresskraft des Schneidrads oder die Vortriebsgeschwindigkeit visualisieren.

Die Thesis ist in 14 Kapitel aufgeteilt. Zunächst werden das Unternehmen, die Herrenknecht AG, und die Kundenplattform *CONNECTED*, die im Rahmen des Projektes erweitert wird, vorgestellt. Daraufhin folgt eine ausführliche Projektbeschreibung, wobei die Ausgangslage sowie die Zielsetzung beschrieben werden. Es folgen zwei theoretische Kapitel zur Datenübertragung im Internet und zum Media-Streaming. Dabei soll ein Blick auf das allgemeine Prinzip des Streamings, die Entwicklung des

Streamings und verschiedene Streaming-Protokolle geworfen werden. Diese Kapitel dienen als Grundlage für die Marktübersicht verschiedener Streaming-Komponenten in Kapitel 6.

Im nächsten Schritt werden Lösungsansätze für das Projekt vorgestellt (Kapitel 7) und anhand gewählter Kriterien verglichen. Schließlich geht es um die Implementierung des gewählten Lösungsansatzes, zunächst auf einem lokalen Rechner und schließlich auf einem Testsystem. Kapitel 10 umfasst die Erweiterung der gewählten Lösungsarchitektur. Im Anschluss wird der entwickelte Prototyp getestet und in Kapitel 13 ein Ausblick gegeben. Dabei soll auch ein Blick in die Zukunft geworfen und auf mögliche Weiterentwicklungen eingegangen werden. Im letzten Kapitel dieser Thesis werden alle Erkenntnisse und die erbrachten Leistungen zusammengefasst.

2. Das Unternehmen – die Herrenknecht AG

Diese Thesis wurde bei der Herrenknecht AG absolviert. In diesem Kapitel werden das Unternehmen und die Produkte vorgestellt. Außerdem werden die Abteilung „Digitale Produkte und Services“ und die Webapplikation *CONNECTED* präsentiert.

2.1. Allgemeines

Die Herrenknecht AG ist eines der führenden Unternehmen im Bereich des maschinellen Vortriebs von Tunnelinfrastrukturen und „der Erschließung unterirdisch gelagerter Energiequellen und Rohstoffvorkommen“ [4]. Gegründet wurde das Unternehmen als Herrenknecht GmbH im Jahr 1977 von Martin Herrenknecht. Eine erste Tochtergesellschaft folgte bereits 1984 in Sunderland, England. In den Jahren 1988 bis 1997 wuchs das Unternehmen stetig weiter. Ein erster Weltrekord wurde für das mit 14,20 Metern Durchmesser weltweit größte Mixschild aufgestellt, das von Herrenknecht entworfen und gebaut wurde [5]. Im Jahr 2006 folgte ein neuer Durchmesser-Weltrekord mit 15,43 Metern [6].

Ende des Jahrtausends erreichte das Unternehmen schließlich die Marktführerschaft in der maschinellen Vortriebstechnik [5]. In den Jahren 2008 bis 2017 wurde die Herrenknecht AG um die Geschäftsfelder Exploration und Mining erweitert. Außerdem erfolgte ein neuer Weltrekord mit einem Mixschild mit einem Durchmesser von 17,6 Metern, das für den Bau eines Straßentunnels in Hongkong eingesetzt wurde (siehe Abbildung 1).

Abbildung 1: weltgrößte Tunnelbohrmaschine, Quelle: [7]

Gegenwärtig arbeiten etwa 5000 Mitarbeiterinnen und Mitarbeiter, verteilt über 30 Länder, bei der Herrenknecht AG [8]. Der Konzern besteht nicht nur aus der Muttergesellschaft, sondern auch aus etwa 70 Tochter- und Beteiligungsgesellschaften [4].

2.2. Geschäftsbereiche

Die Herrenknecht AG umfasst fünf unterschiedliche Geschäftsbereiche, darunter Traffic Tunneling, Utility Tunneling, Exploration, Mining und Group Brands [4].

Tunneling ist das Kerngeschäft der Herrenknecht AG. Dieser Geschäftsbereich umfasst das Traffic Tunneling sowie das Utility Tunneling. Das Traffic Tunneling bietet Vortriebsmaschinen mit Durchmessern von 4,80 bis zu 19 Metern [9], die eingesetzt werden, um Verkehrstunnel für Straße, Eisenbahn und Metro zu erstellen.

Abbildung 2: EPB-Schild, Quelle: [10]

Utility Tunneling umfasst dahingegen Maschinen, die genutzt werden, um Ver- und Entsorgungsinfrastrukturen zu schaffen [11]. Dazu gehören auch Projekte im Bereich der Wasserkraft und Pipelines für Öl und Gas [9]. Dabei geht es um die Erstellung von „Tunnelbauwerke[n] für Wasser, Abwasser, Elektrizität und Kommunikation auf allen Kontinenten“ [11].

Der Geschäftsbereich Mining beinhaltet „Technologien, um den Bau von unterirdischer Infrastruktur in Bergwerken sicherer und effizienter zu gestalten“ [12]. Dazu zählen senkrechte Zugangs- oder Produktionsschächte, schräge Zufahrtsrampen, Belüftungsschächte und Transportstrecken in Tiefen von bis zu 2000 Metern [12].

Zum Geschäftsfeld Exploration gehören Bohranlagen, „um Energiequellen sicher und schnell zu erschließen“ [13]. Dabei ist die Tochtergesellschaft Herrenknecht Vertical für Tiefbohranlagen im On- und Offshore-Bereich verantwortlich, die bis in Tiefen von 8000 Metern vordringen können.

Unter Group Brands werden „individuelle Lösungen rund um den Tunnelvortrieb“ [4] zur Ergänzung der Tunnelbohrmaschinen verstanden. Dazu zählen Zusatzequipment wie Navigations- und Monitoringsysteme, Förderbandsysteme oder Abbauwerkzeuge [14].

2.3. Abteilung „Digitale Produkte und Services“

Diese Thesis wurde in Zusammenarbeit mit der Abteilung „Digitale Produkte und Services“ der Herrenknecht AG geschrieben. Die Abteilung ist aufgeteilt in zwei Teams: „Digitale Produkte“ und „Digitale Services“.

Das Team „Digitale Services“ beschäftigt sich unter anderem mit der Digitalisierung verschiedener Abläufe und Prozesse rund um den Tunnelbau. Dazu gehört auch die Integration künstlicher Intelligenz.

Dahingegen ist das Team „Digitale Produkte“ zuständig für die Kundenplattform *CONNECTED*, die im Rahmen dieser Thesis um das Videostreaming ergänzt werden soll. Diese wird im nächsten Abschnitt detaillierter vorgestellt.

2.4. Webapplikation *CONNECTED*

CONNECTED ist das von Herrenknecht entwickelte Kundenportal, das nicht nur für Kunden, sondern auch für den internen Gebrauch entwickelt wird. Es dient dazu, einen Überblick über die gegenwärtig aktiven Tunnelbohrmaschinen zu liefern [15].

CONNECTED bietet dazu verschiedene Ansichten und Dashboards, die diverse Daten der Maschinen darstellen. Jede Tunnelbohrmaschine verfügt über eine Vielzahl an Sensoren, die regelmäßig Werte liefern. Diese werden in *CONNECTED* in Diagrammen visualisiert, um Projektbeteiligten Einsicht in den Stand der Maschinen zu liefern. So sind unter anderem der aktuelle Standort, die Vortriebsgeschwindigkeit und der Schneidradantrieb dargestellt [15].

Außerdem werden in regelmäßigen Abständen Reporte generiert, die Informationen über die Navigation, den Vortrieb und mögliche Warnungen enthalten [15]. Folgende Abbildung zeigt eine Beispielübersicht des Tunnelfortschritts für eine Demo-Maschine.

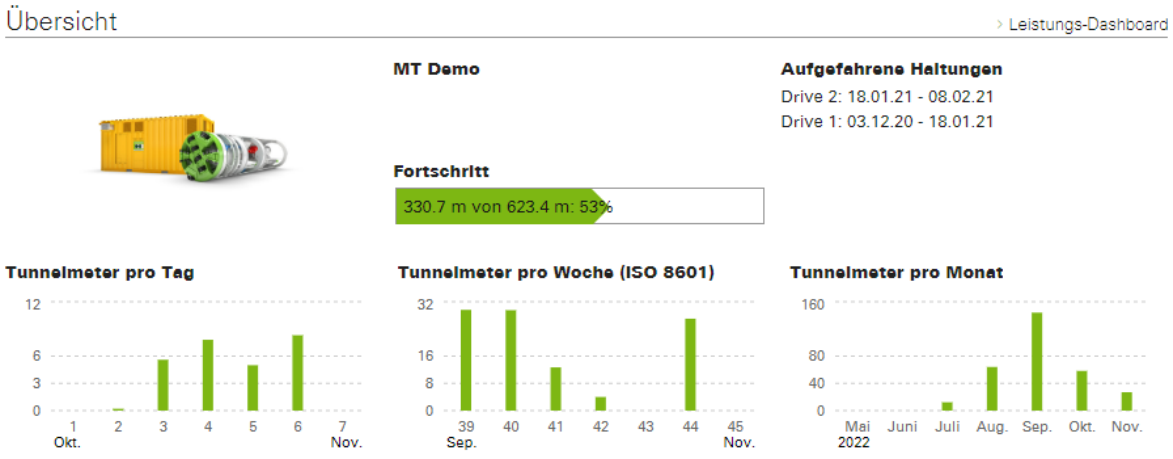

Abbildung 3: Übersicht Demo-Maschine in CONNECTED, Quelle: [16]

3. Das Projekt

Das Thema dieser Thesis lautet „Camera Stream Solution – Marktübersicht, Lösungsansätze, Prototyp“. Dabei geht es um den Transport von Videostreams von den Herrenknecht-Tunnelbohrmaschinen in die Cloud und über das Web-Portal *CONNECTED* zum Browser der Kunden. Das Ziel ist es, dass die Kunden einen Stream des Bildschirminhalts der Rechner auf ihren Tunnelbohrmaschinen im Browser ansehen können, ohne sich auf der Tunnelbohrmaschine zu befinden.

Die gegenwärtige Lösung ermöglicht es bereits, Standbilder bereitzustellen, die im Abstand von zwei bis drei Sekunden abgerufen werden. Im Rahmen der Thesis geht es darum, Bildschirminhalte und in der Zukunft auch Kamerasignale als Videostream mit 20 bis 30 Bildern pro Sekunde anzubieten.

3.1. Ausgangslage

CONNECTED bietet Kunden bereits eine Ansicht, die Standbilder der Rechner auf den Tunnelbohrmaschinen darstellt. Bei den Standbildern handelt es sich vor allem um Cockpit-Monitore der Maschinenfahrer. Dies können dabei Navigationsbildschirme beziehungsweise Visualisierungen relevanter Parameter zur Steuerung der Tunnelbohrmaschine sein. Folgende Abbildung zeigt ein Bespielbild des Navigationsbildschirms einer Demo-Maschine:

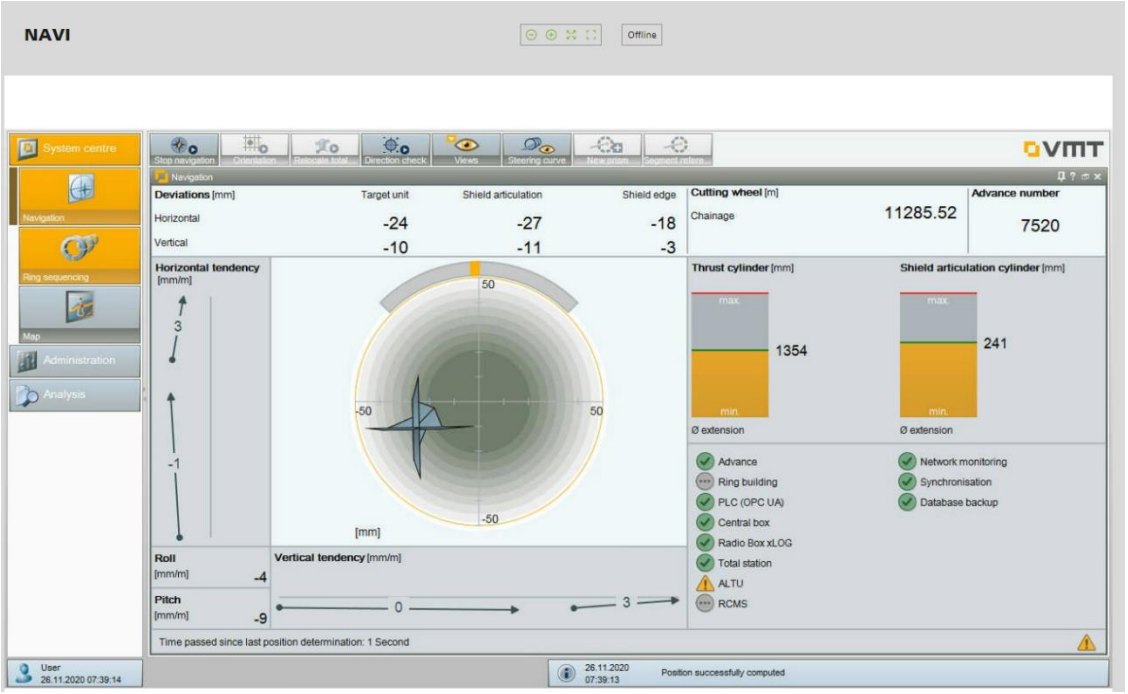

Abbildung 4: Demo-Navigation, Quelle: [17]

Diese Ansicht visualisiert die aktuelle Position der Maschine. Andere Ansichten, die in *CONNECTED* dargestellt sind, sind zum Beispiel Visualisierungen wichtiger Parameter, wie die Anpresskraft des Schneidrads oder die Vortriebsgeschwindigkeit.

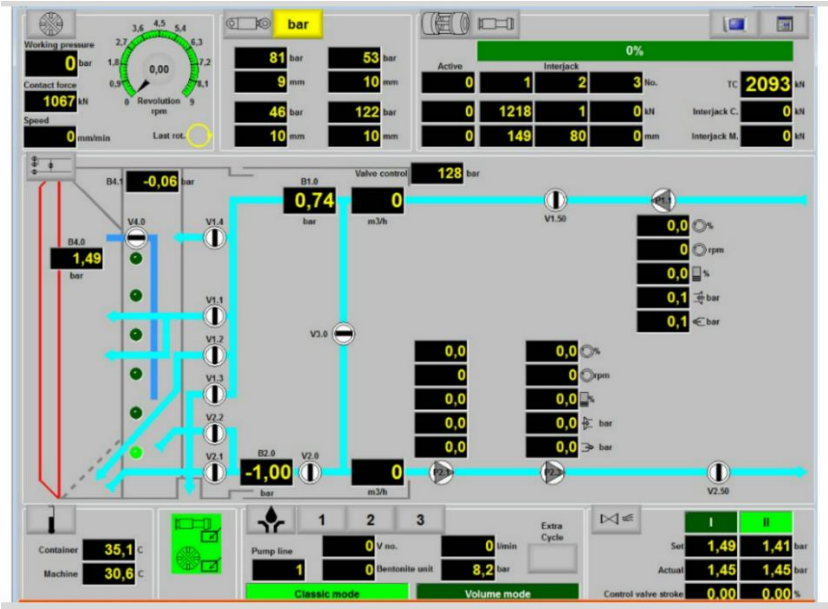

Abbildung 5: Demo-Visualisierung, Quelle: [18]

Die Bereitstellung dieser Daten ist wichtig für Kunden, sodass Projektbeteiligte, die sich nicht auf der Tunnelbohrmaschine befinden, einen Einblick in die Position der Maschine sowie wichtige Parameter haben.

Gegenwärtig werden die Bildschirminhalte in Form von Screenshots alle zwei bis drei Sekunden übertragen und angezeigt. Dabei wird die Übertragung der Screenshots, das als ‚Streaming' bezeichnet wird, gestartet, indem der Nutzer auf den Button „Starte Streaming" klickt. Die Übertragung ist dabei nur möglich, wenn die Maschine online ist, also eine Verbindung hergestellt werden kann.

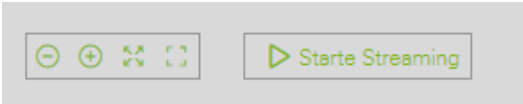

Abbildung 6: Starte Streaming - Button auf CONNECTED, Quelle: [17]

Es handelt sich hierbei noch nicht um echtes Streaming, sondern lediglich um die Übertragung von Standbildern. Folgende Abbildung zeigt die Architektur der Ausgangslage:

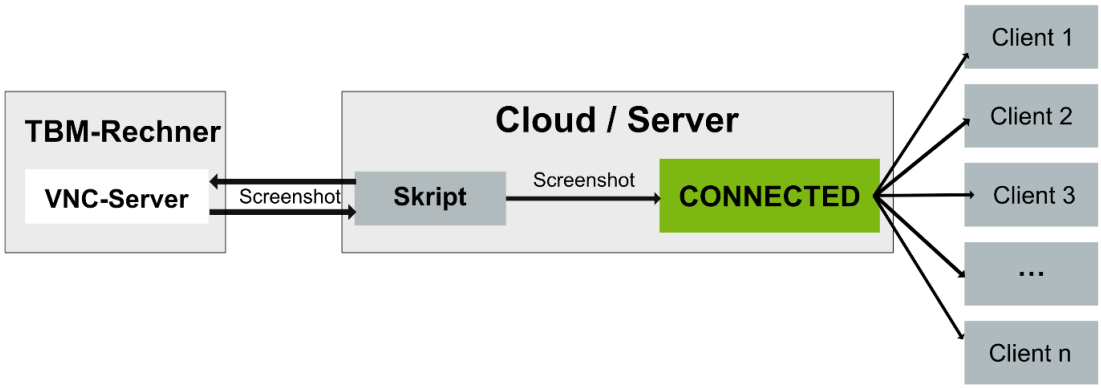

Abbildung 7: Architektur des aktuellen Standes, eigene Darstellung

Die Übertragung läuft ab, indem zunächst eine Verbindung zwischen dem Rechner auf der Tunnelbohrmaschine (TBM) und dem Server, auf dem *CONNECTED* läuft, aufgebaut wird. Um diese Verbindung herzustellen, wird das sogenannte ‚Virtual Network Computing' (VNC) eingesetzt. VNC ist eine open-source-Technologie, die es ermöglicht, den Bildschirminhalt eines Rechners über jedes Netzwerk zu übertragen. Dadurch können entfernte Rechner betrachtet, aber auch gesteuert werden [19].

Um eine Verbindung zwischen zwei Rechnern herzustellen, ist einerseits ein VNC-Server auf dem entfernten Rechner und andererseits ein VNC-Client oder VNC-Viewer auf dem anderen Desktop notwendig.

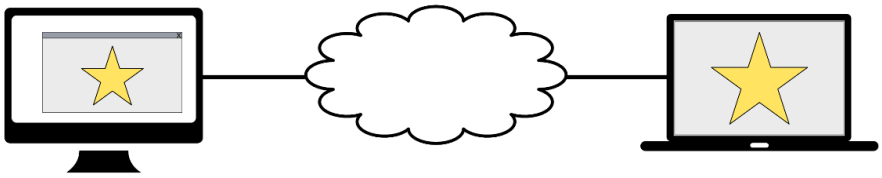

Abbildung 8: VNC-Prinzip, eigene Darstellung in Anlehnung an [19]

Die VNC-Technologie basiert auf dem *Remote-Framebuffer*-Protokoll, das diese Steuerung ermöglicht [20]. Dabei handelt es sich um ein einfaches Protokoll, wobei der VNC-Server lediglich Bildschirmänderungen in Form von Pixeldaten überträgt, die im nächsten Schritt vom VNC-Client dargestellt werden [20]. Die Übertragung wird vom VNC-Client gesteuert, indem dieser ein sogenanntes ‚*Framebuffer Update*' anfordert und im nächsten Schritt die Änderungen vom Server erhält. Dadurch ist die VNC-Technologie adaptiv und die Update-Rate kann in Abhängigkeit von der Netzwerkgeschwindigkeit angepasst werden [20].

Die Herrenknecht AG nutzt die VNC-Technologie, indem auf den Rechnern in den Tunnelbohrmaschinen jeweils ein VNC-Server läuft. Auf dem Server in der Cloud ist ein Skript installiert, das eine Verbindung zum VNC-Server auf dem TBM-Rechner aufbaut und Screenshots vom übertragenen Inhalt macht (siehe Abbildung 7). Ein VNC-Client ist demnach nicht notwendig. Die Screenshots des übertragenen Inhalts werden etwa alle zwei bis drei Sekunden durch das Skript gemacht und schließlich in *CONNECTED* dargestellt. Die Auflösung der Screenshots beträgt dabei meistens Full HD, also 1920 x 1080 Pixel, kann aber teilweise auch 3840 x 2160 Pixel betragen.

3.2. Erweiterung und Zielsetzung

Das Ziel dieses Projekt ist es, die Darstellung von Screenshots mit einer echten Streaming-Lösung zu ersetzen. Das bedeutet, dass ein Videostream des Bildschirminhaltes mit 20 bis 30 Bildern pro Sekunde in *CONNECTED* zur Verfügung gestellt werden soll.

Zu den Anforderungen gehört es, zu vermeiden, dass Clients auf direkte Weise eine Verbindung, zum Beispiel über VNC, zu den TBM-Rechnern aufbauen. Dies birgt ein großes Risiko, da Clients dadurch möglicherweise Zugriff auf diese Rechner bekommen könnten. Aus Sicherheitsgründen soll es vermieden werden, dass Clients über die VNC-Server Zugriff auf die TBM-Rechner erhalten. Hinzu kommt, dass der Datenverkehr auf der Leitung in der Tunnelbohrmaschine so gering wie möglich gehalten werden soll. Dementsprechend ist eine weitere Anforderung, dass der Bildschirminhalt als Stream nur einmal vom TBM-Rechner in die Cloud transportiert werden soll. In der Cloud soll eine Software eingesetzt werden, die diesen Videostream entgegennimmt und an alle Clients, also Web-Browser, verteilt. Die Auflösung soll mindestens Full HD, also 1920 x 1080 Pixel, oder, wenn dies möglich ist, 3840 x 2160 Pixel betragen. Dabei ist ebenfalls wichtig, die Verzögerungszeit (Latenz) bei der Übertragung zwischen TBM-Rechner und Clients so gering wie möglich zu halten. Das Ziel ist es, so nah wie möglich an eine Echtzeit-Übertragung zu kommen.

Die Aufgaben des Projektes umfassen die Recherche rund um das Thema Media-Streaming und die Bereitstellung einer Marktübersicht verschiedener Streaming-Komponenten. Zudem sollen mögliche Lösungsansätze in Bezug auf das Projekt ermittelt und verglichen werden und es soll ein Ansatz anhand bestimmter Kriterien für das Projekt ausgewählt werden. Weiterhin soll eine geeignete Struktur aufgebaut und ein Prototyp basierend auf dem gewählten Lösungsansatz implementiert werden. Letztlich wird eine Evaluation durchgeführt und mögliche Erweiterungen und Optimierungen für die Zukunft werden betrachtet.

4. Datenübertragung im Internet

In diesem Kapitel geht es um die Datenübertragung im Internet und es wird die Grundlage zum Verständnis der darauffolgenden Kapitel zum Media-Streaming geschaffen.

4.1. Allgemeines Prinzip der Datenübertragung

Die Datenübertragung innerhalb von Computernetzen geschieht entweder über die sogenannte ‚Leitungsvermittlung' (*Circuit-Switched Networking*) oder über die Paketvermittlung (*Packet-Switched Networking*) [21]. Die Leitungsvermittlung stammt aus der Telefonie und basiert auf dem Aufbau von Ende-zu-Ende-Verbindungen [21]. Die Kommunikation läuft dann über diese Verbindung. Dahingegen werden Daten bei der Paketvermittlung in mehrere Pakete verpackt und einzeln über das Netzwerk übertragen [21]. Dabei wird jedes Paket über verschiedene Knoten im Netzwerk von einem Rechner zu einem anderen übertragen. Jedes Paket kann verschiedene Routen innerhalb des Netzes zum Ziel nehmen. Der Empfänger setzt die Pakete schließlich wieder zusammen [21].

Damit die Pakete den richtigen Empfänger erreichen, werden Protokolle eingesetzt, die sich in den sogenannten ‚Paket-Headern' befinden. „Ein Protokoll ist eine Menge von Regeln, die das Verhalten von Instanzen oder Prozessen bei der Kommunikation festlegt." [22] Sie liefern Informationen zu den ausgetauschten Daten und sind vor allem für die Adressierung von Sender und Empfänger zuständig [21]. Die eingesetzten Protokolle sind auf verschiedenen Ebenen angeordnet. Dazu wurden verschiedene Schichtenmodelle entwickelt, wobei jede Schicht einen bestimmten Teil der Kommunikation übernimmt [23]. Außerdem bietet jede Schicht Schnittstellen zur darüberliegenden und darunterliegenden Schicht. Eines der bekanntesten Modelle ist das sogenannte ‚ISO/OSI-Referenzmodell', das im nächsten Abschnitt vorgestellt wird.

4.2. Das ISO/OSI-Referenzmodell

Das ISO/OSI-Referenzmodell wurde im Jahr 1983 von der ISO (Internationale Organisation für Normung) standardisiert. „OSI steht für *Open Systems Interconnection*.“ [21] Es definiert sieben verschiedene Schichten, die in Abbildung 9 dargestellt sind und jeweils unterschiedliche Aufgaben besitzen. Dabei erfüllt jede Schicht einen Dienst, wobei Protokolle die Kommunikation innerhalb einer Schicht steuern. Nachrichten werden von Schicht zu Schicht weitergeleitet und gekapselt [24].

Abbildung 9: ISO/OSI-Referenzmodell, eigene Darstellung in Anlehnung an [21]

Auf unterster Ebene befindet sich die Bitübertragungsschicht, die für die eigentliche Übertragung der einzelnen Bits über ein bestimmtes Medium, wie Kabel oder Luft, zuständig ist. Auf dieser Schicht werden die Daten-Bits codiert und teilweise um Redundanzen erweitert, um Bitfehler auszugleichen [21].

Die Sicherungsschicht auf zweiter Ebene umfasst die Übertragung von Bitblöcken zwischen Netzteilnehmern. Diese werden auch als ‚Datenrahmen‘ bezeichnet und über eine minimale und maximale Länge beschrieben. Auf dieser Ebene wird nicht nur der Bitstrom in Datenrahmen aufgeteilt, sondern diese werden auch gegen Fehler abgesichert und der Datenfluss wird gesteuert [21]. Außerdem werden die Endgeräte physikalisch adressiert [21].

Die dritte Schicht wird als ‚Vermittlungsschicht‘ bezeichnet und ist für die „*logische Adressierung* von Geräten über Netzgrenzen hinweg und die *Fragmentierung* von zu großen Datagrammen“ [21] zuständig. Das bekannteste Protokoll der Vermittlungsschicht ist das Internet-Protokoll (IP) [21].

Über der Vermittlungsschicht befindet sich die Transportschicht und ist dafür verantwortlich, Verbindungen auf- und abzubauen und Datagramme zu übertragen [21]. Zu den Transportprotokollen zählen zum Beispiel das *User Datagram Protocol* (UDP) und das *Transmission Control Protocol* (TCP).

Die fünfte Schicht wird im ISO/OSI-Referenzmodell als ‚Sitzungsschicht‘ bezeichnet [21]. Sie ist für Verbindungen oder Sitzungen verantwortlich und stellt sicher, dass mehrere Nutzer nicht gleichzeitig „auf kritische Operationen“ [21] zugreifen, und bestimmt, wer zu welchem Zeitpunkt übertragen darf.

Auf der nächsten Schicht, der Darstellungsschicht, werden die übertragenen Daten codiert und decodiert. Diese Schicht ist zudem für die Kompression und Verschlüsselung verantwortlich [21].

Letztlich ist die Anwendungsschicht dafür verantwortlich, mit der Anwendung, also zum Beispiel einem Web-Browser oder E-Mail-Client, zu kommunizieren. Die Anwendungsschicht greift auf die Dienste der darunterliegenden Schichten zu [21].

4.3. Das 4-Schichten-Internetmodell

Das ISO/OSI Modell wurde ursprünglich dazu entwickelt, die Funktionsweise und die Kommunikation von Netzwerken zu visualisieren [25]. Allerdings wird das Modell mittlerweile von vielen Wissenschaftlern als veraltet betrachtet [25, 26, 27]. Stattdessen kommen alternative Modelle wie das 4-Schichten-Internetmodell beziehungsweise TCP/IP-Modell zum Einsatz, das im Folgenden vorgestellt wird.

Im Gegensatz zum ISO/OSI-Modell besteht das 4-Schichten-Internetmodell, auch TCP/IP-Model genannt, lediglich aus vier Schichten und einer Sammlung an Protokollen, die bestimmen, wie Daten über das Netzwerk transportiert werden [28]. Es wurde vom *Department of Defense* (DoD) entwickelt, um mehrere Geräte an ein gemeinsames Netzwerk anzuschließen [28]. TCP/IP ist die Abkürzung für ‚Transmission Control Protocol/Internet Protocol', wobei die Protokolle TCP und IP die Hauptkomponenten des Modells ausmachen. Folgende Abbildung zeigt die vier Schichten dieses Modells:

Abbildung 10: TCP/IP-Modell, eigene Darstellung in Anlehnung an [28]

Die Anwendungsschicht im TCP/IP-Modell entspricht der Anwendungs-, Präsentations- und Sitzungsschicht im ISO/OSI-Modell. Anwendungen auf der Anwendungsschicht sind beim TCP/IP-Modell dafür zuständig, Nutzerdaten zu erstellen und diese Daten zwischen anderen Anwendungen auf demselben oder einem anderen Host auszutauschen [28]. Die meistgenutzten Protokolle dieser Schicht sind unter anderem HTTP, FTP und SMTP [28].

Die Transportschicht entspricht derselben Schicht im ISO/OSI-Referenzmodell. Diese Schicht ermöglicht Ende-zu-Ende-Kommunikation zwischen Hosts [28]. Dabei kommen Transportprotokolle, wie UDP und TCP, zum Einsatz, auf die im weiteren Verlauf dieses Kapitels näher eingegangen wird.

Die Internetschicht (Schicht 2) ist dafür zuständig, Datagramme, also Datenpakete, über Netzwerkgrenzen hinweg zu übertragen [28]. Diese Schicht definiert unter anderem, wie das Routing und die Adressierung von Netzwerkteilnehmern geschehen. Das Hauptprotokoll, das hier Einsatz findet, ist das Internet-Protokoll (IP). Dieses definiert IP-Adressen, die in Paketen mitgesendet werden, um den Sender sowie den Empfänger zu adressieren [28].

Letztlich entspricht die Netzwerkschicht der Sicherungs- und Bitübertragungsschicht im ISO/OSI-Modell. Sie bildet die Schnittstelle zwischen Host und Netzwerk. Dabei können zum Beispiel Technologien wie Ethernet und LAN zum Einsatz kommen [28].

Generell wird das TCP/IP-Modell als praktischer angesehen als das ISO/OSI-Referenzmodell. Wie bereits erwähnt wurde, wird das ISO/OSI-Referenzmodell teilweise als veraltet angesehen, wofür es verschiedene Gründe gibt. Zum einen ist das TCP/IP-Modell flexibler und dadurch besser an neue Technologien anpassbar [25]. Außerdem sind die einzelnen Schichten im Gegensatz zum ISO/OSI-Modell weniger stark voneinander abhängig [25]. Ein weiteres Argument für das TCP/IP-Modell ist, dass das ISO/OSI-Modell auf die sogenannte ‚Full-Stack-Entwicklung', also die Front- und Backend-Entwicklung, ausgelegt ist, wohingegen neuere Modelle es ermöglichen, einzelne Komponenten zu integrieren, was sie effizienter macht [25].

4.4. Transportprotokolle

Auf der Transportschicht werden Daten auf Senderseite in Segmente verpackt und Prozesse mit Portnummern adressiert [23]. Beim Empfänger können die Segmente in den Paketen schließlich mithilfe der Transportschicht erkannt werden, wofür verschiedene Transportprotokolle eingesetzt werden. Zwei bekannte Protokolle sind das *Transmission Control Protocol* (TCP) und das *User Datagram Protocol* (UDP). Da diese für das Streaming von großer Bedeutung sind, werden sie im Folgenden detaillierter vorgestellt.

TCP [21]	**UDP** [21]
zuverlässig	unzuverlässig
verbindungsorientiert	verbindungslos
reihenfolgetreu (Sequenznummern)	nicht reihenfolgetreu (keine Sequenznummern)
für verlustempfindliche Anwendungen	für verlusttolerante Anwendungen

Tabelle 1: TCP vs. UDP, eigene Darstellung

4.4.1. Transmission Control Protocol (TCP)

TCP ist ein verbindungsorientiertes und zuverlässiges Transportprotokoll [21]. Dies bedeutet, dass es virtuelle Verbindungen zwischen Prozessen aufbaut und für einen sicheren Datenaustausch zwischen diesen sorgt [29]. Folgende Abbildung zeigt den Verbindungsaufbau mit TCP zwischen zwei Hosts, der auch als ‚*3-Way-Handshake*' bezeichnet wird [29].

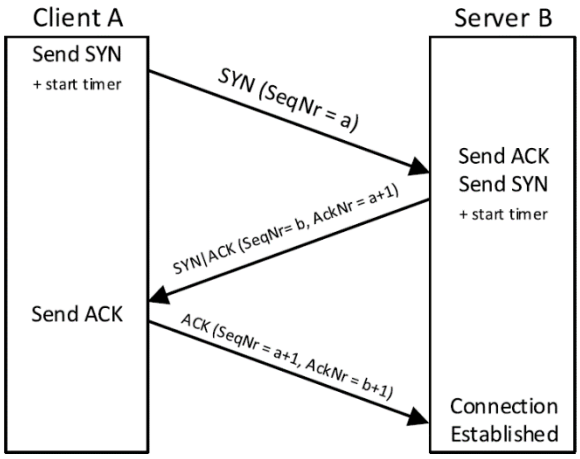

Abbildung 11: TCP-Verbindungsaufbau, Quelle: [21]

In einem ersten Schritt sendet der Client ein Paket an den Server beziehungsweise den Host, mit dem eine Verbindung aufgebaut werden soll. Darin ist das sogenannte ‚SYN-Flag' gesetzt, was indiziert, dass ein Verbindungsaufbau gewünscht ist [29]. Außerdem erhält das Paket eine zufällige Sequenznummer. Als nächstes sendet der Server eine Bestätigung der Anfrage mit einem gesetzten SYN- und ACK-Flag. Dabei ist nicht nur eine Sequenznummer angegeben, sondern auch eine sogenannte ‚*Acknowledgement Number*', womit die Sequenznummer des initiierenden Clients bestätigt und diesem mitgeteilt wird, welche Sequenznummer als nächstes erwartet wird [29]. Durch die Kombination dieser Nummern wird eine bidirektionale Verbindung aufgebaut. Zuletzt sendet der initiierende Client eine Bestätigung der Sequenznummer und die vom Server erwarteten Bytes [29]. Während des Verbindungsaufbaus werden auch andere Daten wie zum Beispiel die Portnummer des aufgerufenen Dienstes mitgesendet [29].

Nach dem Verbindungsaufbau können Daten schließlich ausgetauscht werden, wobei Daten in Pakete zerlegt und ins Netz geschickt werden. Der Empfänger speichert die eintreffenden Pakete in einem Puffer, bevor die entsprechende Anwendung darauf zugreift [29]. Nach der Übertragung wird die Verbindung zwischen den Prozessen wieder beendet.

Neben den bereits erwähnten Sequenznummern, die sicherstellen, dass der Empfänger eintreffende Pakete in die korrekte Reihenfolge sortieren kann, verfügt TCP über weitere Kontrollmechanismen, die einen sicheren Datenaustausch gewährleisten. Dazu zählen die Stau- und die Flusskontrolle [21].

Bei der Staukontrolle beziehungsweise -vermeidung setzt TCP den sogenannten ‚Slow-Start-Algorithmus' ein, um die höchstmögliche Senderate zu ermitteln, ohne dass Stau auftritt [21]. Dabei beginnt der Sender mit einer bestimmten Senderate und verdoppelt nach jeder erfolgreichen Übertragung, die durch eine Bestätigung des Empfängers indiziert wird, die übertragenen Pakete [21]. Wird ein bestimmter Schwellenwert erreicht, steigt die Senderate nur noch linear. Tritt ein Fehler auf, was der Sender daran erkennt, dass innerhalb eines bestimmten Zeitraums keine Bestätigung eintrifft, wird die Senderate stark gesenkt. Mit diesem Prinzip wird versucht, eine Überlastsituation zu vermeiden beziehungsweise im Fall der Überlastung die Senderate anzupassen [21].

Die Flusskontrolle sorgt dafür, dass die „Leitungs- und Empfangskapazität besser ausgelastet" [23] wird. Dabei nutzt TCP das sogenannte ‚*Sliding-Window*-Verfahren', um eine bestimmte Anzahl an Segmenten zu senden, bevor eine Quittung (Bestätigung) des Empfängers erwartet wird. Sobald der Empfänger ein Paket oder mehrere Pakete bestätigt, wird das Fenster verschoben und neue Segmente werden versendet [23]. Erhält der Sender in einem bestimmten Zeitraum keine Quittung, sendet dieser alle nichtquittierten Pakete erneut. Die Fenstergröße, also wie viele Segmente versendet werden können, wird durch die Größe des Pufferspeichers auf Empfängerseite bestimmt [23]. Dies verhindert Pufferüberläufe.

Wie im Verlauf dieser Thesis näher betrachtet wird, findet TCP auch für das Streaming Anwendung. Dennoch muss beachtet werden, dass TCPs Kontrollmechanismen zu längerer Verzögerung und Jitter, also Schwankungen in der Übertragung, führen können, was sich auf den Stream auswirkt, indem das Video einfriert oder neu gepuffert wird [30]. Auf mögliche Alternativen wird im Verlauf dieser Thesis eingegangen.

4.4.2. User Datagram Protocol (UDP)

Das *User Datagram Protocol* (UDP) ist ein Netzwerkprotokoll, das eine verbindungslose und unzuverlässige Übertragung ermöglicht [21]. Dies bedeutet, dass es im Vergleich zu TCP keine Verbindung zwischen den Kommunikationsteilnehmern aufbaut. Außerdem verfügt UDP weder über Sequenznummern noch über Kontroll- und Regulierungsmechanismen des ausgehenden Datenstroms [21], weshalb UDP keine reihenfolgegetreue Übertragung von Paketen bietet.

Durch die fehlenden Kontrollmechanismen kann UDP in Überlastsituationen die Senderate nicht reduzieren. Wenn in solch einem Fall Pakete verloren gehen, bietet UDP keine Möglichkeit, dieses erneut zu übertragen [21]. Aus diesem Grund wird UDP häufig für zeitkritische Aufgaben wie die Echtzeitkommunikation oder andere verlusttolerante Anwendungen eingesetzt [23]. TCP puffert die eingehenden Pakete und wartet auf beim Paketverlust auf die erneute Übertragung der entsprechenden Pakete. Allerdings führt diese Pufferung zu Verzögerungen, die, insbesondere beim Streaming oder anderen Echtzeitanwendungen, verhindert werden sollen [23]. Bei UDP werden Pakete nicht erneut

übertragen, was sich bei Paketverlust in Echtzeitanwendungen lediglich auf ein Bild oder ein Segment auswirkt.

Aus genannten Gründen ist UDP besonders gut für das Streaming geeignet, da es ohne diverse Kontrollmechanismen geringere Verzögerungszeiten bietet [30]. Allerdings sind einige Videokompressionsverfahren auf aufeinanderfolgende Frames angewiesen, sodass es bei Paketverlust zu einer fehlerhaften Rekonstruktion des Signals kommen kann [30]. Dennoch wird UDP für einige Streaming-Anwendungen eingesetzt.

Im Verlauf dieser Thesis wird insbesondere noch ein Blick auf Streaming-Protokolle wie RTMP, HLS und MPEG-DASH geworfen.

5. Streaming-Grundlagen

Streaming, auch als ‚Media Streaming' oder ‚Real Time Multimedia' bezeichnet, ist ein Prozess, bei dem digitale Medien kontinuierlich von einer Quelle aus an einen oder mehrere Empfänger übertragen werden [31]. Dabei werden die Daten, im Gegensatz zu lokal gespeicherten Dateien, bereits während der Übertragung wiedergegeben [31]. Das bedeutet, dass zum Beispiel ein Video progressiv heruntergeladen wird und gleichzeitig bereits ausgespielt werden kann [32].

Gegenwärtig wird Streaming für verschiedene Anwendungen wie zum Beispiel Fernunterricht, Online-Videokonferenzen, Internet-TV oder On-Demand-Angebote eingesetzt [2]. Dabei ist es wichtig, dass die Daten in Echtzeit geliefert und gerendert werden. Meist wird eine große Bandbreite benötigt, um sicherzustellen, dass die Daten mit möglichst geringer Verzögerung beim Empfänger ankommen [2]. Kommen die Daten nicht rechtzeitig an, um das Video oder Audio kontinuierlich abspielen zu können, wird die Wiedergabe gestoppt. Solche Artefakte sind deutlich von Menschen wahrnehmbar [2]. Um diese zu vermeiden, wird auf Empfängerseite zusätzlich ein Pufferspeicher eingesetzt, der eingehende Datenpakete puffert und mit einer bestimmten Verzögerung wiedergibt, sobald genügend Daten vorhanden sind. Dies ist notwendig, da die Datenpakete unterschiedlich schnell beim Empfänger ankommen [2]. Eine detaillierte Vorstellung des Streaming-Prinzips folgt im Verlauf dieses Kapitels.

Generell wird das echte Streaming von einem Download und einem sogenannten ‚progressiven Download' unterschieden. Während beim Streaming audiovisuelle Inhalte in Echtzeit übertragen und abgespielt werden können, wird bei einem Download zunächst eine Kopie der gesamten Datei auf dem lokalen Endgerät gespeichert. Erst nachdem die Datei vollständig heruntergeladen ist, kann sie abgespielt werden [33]. Dies hat zwar den Vorteil, dass Dateien auch ohne Internetverbindung wiedergegeben werden können, allerdings verbraucht dies mit der Zeit viel Speicherplatz, vor allem bei längeren, größeren Mediendateien. Außerdem birgt das Bereitstellen von Dateien zum Download die Gefahr der unautorisierten Distribution [34].

Der progressive Download ist dahingegen ein hybrider Ansatz zwischen einem Download und Streaming [35]. Beim progressiven Download wird, wie bei einem regulären Download, eine Datei von einem Server heruntergeladen [36]. Allerdings kann die Datei bereits abgespielt werden, während die Datei heruntergeladen wird. Im Gegensatz zum Streaming kann dies jedoch nicht in Echtzeit ablaufen [35]. Beim progressiven Download wird jeweils ein Großteil einer Datei in den Browser-Cache geladen, während der Browser die Datei gleichzeitig abspielt. Die Datei wird also beim progressiven Download im Browser-Cache gespeichert, wohingegen sie beim Streaming direkt nach dem Abspielen wieder gelöscht wird [35]. Außerdem kann der Nutzer beim progressiven Download nicht Vorspulen, sondern muss warten, bis die Datei bis zum gewünschten Zeitpunkt heruntergeladen ist [35]. Der progressive Download bietet zudem keine adaptive Bitrate während der Wiedergabe [37]. Auf das adaptive Bitrate-Streaming wird im Verlauf dieses Kapitel näher eingegangen.

5.1. Allgemeines Prinzip des Streamings

In der Regel werden Streams auf dieselbe Weise wie andere Daten über das Internet übertragen [30]. Das bedeutet, dass Inhalte wie zum Beispiel eine Audiodatei oder ein Video in Datenpakete aufgeteilt und einzeln durch das Netzwerk an den Endnutzer übertragen werden. Folgende Abbildung zeigt die allgemeine Streaming-Pipeline über das Internet in Anlehnung an Jedari et al. [32], die im Folgenden detailliert erläutert wird.

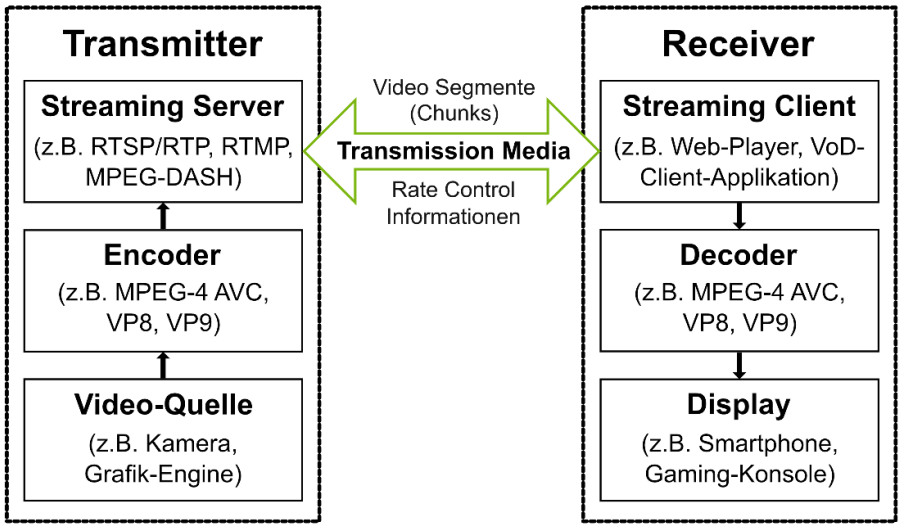

Abbildung 12: Streaming-Pipeline über das Internet, eigene Darstellung in Anlehnung an [32]

Die Pipeline beginnt an erster Stelle mit der Aufnahme der Inhalte, in diesem Fall einem Video. Dies kann durch eine Kamera beziehungsweise mehrere Kameras geschehen oder das Videosignal kann künstlich generiert werden [32]. Im nächsten Schritt wird das Videosignal an den Encoder weitergeleitet, der dafür verantwortlich ist, das Video in einen *Bitstream* zu komprimieren. Dafür werden Codecs eingesetzt. Ein Codec ist ein Algorithmus beziehungsweise eine Methode, mit der Audio- oder Videodaten komprimiert werden. Codecs enthalten einen Encoder sowie einen Decoder [38]. Der Encoder sorgt im ersten Schritt dafür, dass eingehende Signale in ihrer Größe reduziert werden [39], was verlustbehaftet oder verlustfrei geschehen kann. Dabei werden redundante Informationen im Signal so verpackt, dass das Video ressourcenschonender repräsentiert wird [32]. Ist das Videosignal danach 1:1 wiederherstellbar, wird vom ‚*lossless encoding*' gesprochen, also der verlustfreien Komprimierung. Ist dies nicht der Fall, kann es zu Qualitätseinbußen kommen, was auch als ‚*lossy encoding*' bezeichnet wird [32].

Im nächsten Schritt wird der Stream vom Encoder an den Streaming-Server übermittelt. Dieser Schritt wird auch als ‚stream ingestion' oder ‚ingest' bezeichnet [40]. Der Streaming-Server übernimmt den Bitstream und verpackt ihn in eine Form, sodass die Daten über das Internet übertragen werden können. Diese Form wird durch das eingesetzte Streaming-Protokoll bestimmt [32]. Danach transportiert das

Streaming-Protokoll das Video in einzelnen Segmenten, auch ‚*Chunks*' genannt, über das Internet zum Streaming-Client auf der Receiver-Seite [32].

Anschließend wird ein Decoder eingesetzt, um das komprimierte Video, das vom Client erhalten wurde, wieder in das Originalformat zu konvertieren [32]. Abhängig vom eingesetzten Codec kann Qualitätsverlust entstehen. Die Videoqualität hängt aber nicht nur vom Encoding-Schema ab, sondern auch von den Netzwerkbedingungen. Um das Video schließlich abspielen zu können, ist ein Video-Player notwendig, der die dekodierten Videosegmente in einem Puffer sammelt. Die Wiedergabe auf dem Display startet dann mit einer Verzögerung, sobald genügend Inhalte im Puffer gespeichert sind. Der Industriestandard ist derzeit der HTML5 Video-Player, der von fast allen Internet-fähigen Geräten unterstützt wird und mit den meisten Browsern und Betriebssystemen kompatibel ist [41].

Abhängig vom eingesetzten Streaming-Protokoll kann die Übertragungsrate zwischen Transmitter und Receiver dynamisch an die Netzwerkbedingungen angepasst werden [32]. Dies wird auch als ‚*Adaptive Bitrate Streaming*' bezeichnet.

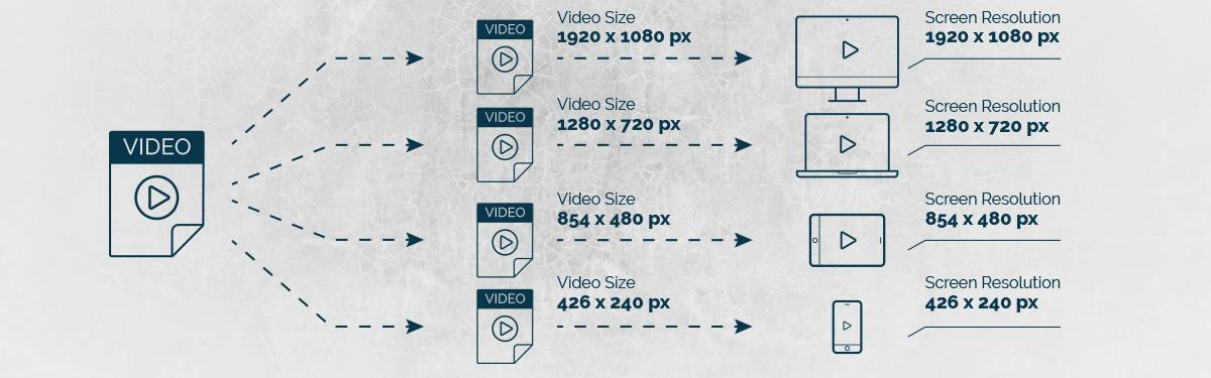

Abbildung 13: Adaptive Bitrate Streaming, Quelle: [42]

Beim Adaptive Bitrate Streaming (ABR-Streaming) wird die Bitrate eines Streams in Abhängigkeit von der verfügbaren Bandbreite, dem Endgerät sowie der Internetgeschwindigkeit angepasst. Das Ziel ist es, das Benutzererlebnis zu verbessern, indem die bestmögliche Qualität eines Streams an den Client geliefert wird [42]. Einfache ABR-Systeme verfügen dabei über einen Streaming-Server und einen Client. Der Streaming-Server speichert dazu die Videosegmente (*Chunks*), die jeweils einige Sekunden lang sind, in verschiedenen Qualitätsstufen ab (siehe Abbildung 13). Das bedeutet, dass die Segmente im Voraus mit unterschiedlichen Bitraten codiert auf dem Server abgelegt werden [43]. Im nächsten Schritt wird eine sogenannte ‚Manifest-Datei' vom Video-Player heruntergeladen, die dem Client Auskunft über die verfügbaren Bitraten und Auflösungen gibt [42]. Der eingesetzte ABR-Algorithmus bestimmt dann die geeignete Bitrate basierend auf der verfügbaren Bandbreite und der CPU-Kapazität des Clients und lädt das Segment in entsprechender Qualität herunter [43]. Das Besondere am ABR-Streaming ist, dass der Video-Player dynamisch die Bitrate anpassen kann, sobald sich die Netzwerkbedingungen ändern [42]. So kann der Client zum Beispiel eine geringere Bitrate anfordern, wenn der Datendurchlauf oder die Datenmenge im Puffer geringer wird. Eine hohe Bitrate bietet zwar

eine bessere Qualität, führt jedoch auch zu einer höheren Netzbelastung. ABR-Streaming ermöglicht es demnach, die Qualität und Netzbelastung zu steuern [43].

Das ABR-Streaming wird zum Beispiel vom Streaming Protokoll HLS (HTTP-Live-Streaming) unterstützt. Auch kommerzielle Streaming-Plattformen wie YouTube und Netflix nutzen diese Technologie [43].

Ein besonders wichtiger Parameter, auf den im Folgenden näher eingegangen wird, ist die sogenannte ‚Latenz'. Sie bestimmt, mit welcher Verzögerung ein Stream übertragen wird, also welche Zeit zwischen der Aufnahme und der Wiedergabe eines Frames vergeht [44]. Für Live-Streaming-Lösungen ist dies der wichtigste Parameter [45].

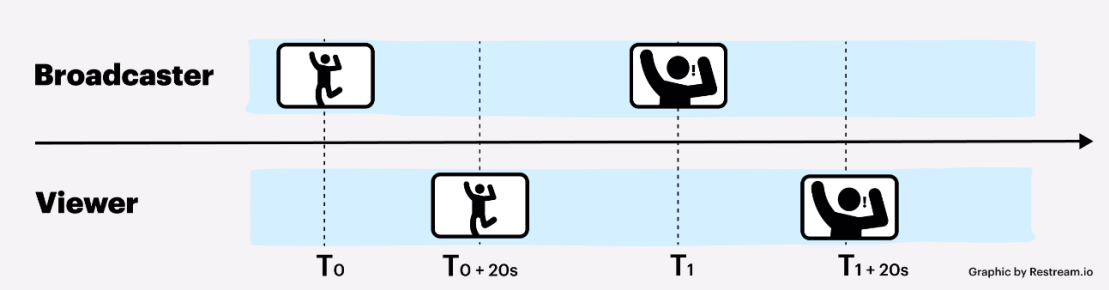

Abbildung 14: Videolatenz, Quelle: [44]

Dabei sind keine Standards definiert, die bestimmen, was als niedrige Latenz (‚low latency') oder hohe Latenz (‚high latency') angesehen wird [44]. Generell sind 30 bis 60 Sekunden Latenz jedoch im höheren Bereich, wenn es um das Online-Video-Streaming geht, während weniger als fünf Sekunden als niedrige Latenz betrachtet werden [44].

Die Latenz hängt von verschiedenen Einflussfaktoren ab. Einmal spielt die Auflösung beziehungsweise das Format des Videostreams eine Rolle. Es gilt: Je größer die Dateien sind, desto länger dauert die Übertragung und desto größer wird die Latenz [44]. Auch die Kodierung des Streams vor der Übertragung kann zur Latenz beitragen. Abhängig davon, wie der Encoder konfiguriert wird, kann dieser eingehende Signale schneller oder langsamer kodieren und übertragen [44].

Ein weiterer Einflussfaktor ist die verfügbare Bandbreite beziehungsweise sind die Netzwerkbedingungen. Je mehr Bandbreite zur Verfügung steht, desto weniger Stau bildet sich im Netzwerk und desto schneller lassen sich Streams über das Internet übertragen [44]. Damit einher geht die Verbindungsart, also ob es sich zum Beispiel um Glasfaser- oder drahtlose Verbindungen handelt. Auch die physische Distanz zwischen Endnutzern und zum Beispiel dem Internet-Service-Provider hat einen Einfluss auf die Latenz [44]. Zuletzt spielt das eingesetzte Streaming-Protokoll eine große Rolle, da es bestimmt, wie Daten von der Quelle zum Endnutzer übertragen werden. Mehr Erläuterungen zu den gängigen Streaming-Protokollen folgen im Verlauf dieses Kapitels.

5.2. On-Demand-Streaming vs. Live-Streaming

Beim Streaming wird zwischen On-Demand- und Live-Streaming unterschieden [46]. Beim sogenannten ‚On-Demand-Streaming' werden die Mediendateien im Voraus aufgenommen und komprimiert. Sie werden auf einem Server gespeichert und an einen oder mehrere Empfänger gesendet, sobald diese die Dateien anfordern [46]. Zu On-Demand-Anbietern gehören unter anderem Netflix, Amazon Prime Video oder YouTube. Folgende Abbildung zeigt den zeitlichen Verlauf beim On-Demand-Streaming. Die x-Achse repräsentiert die vergangene Zeit, während die y-Achse die gesamte übertragene Datenmenge darstellt (siehe Abbildung 15).

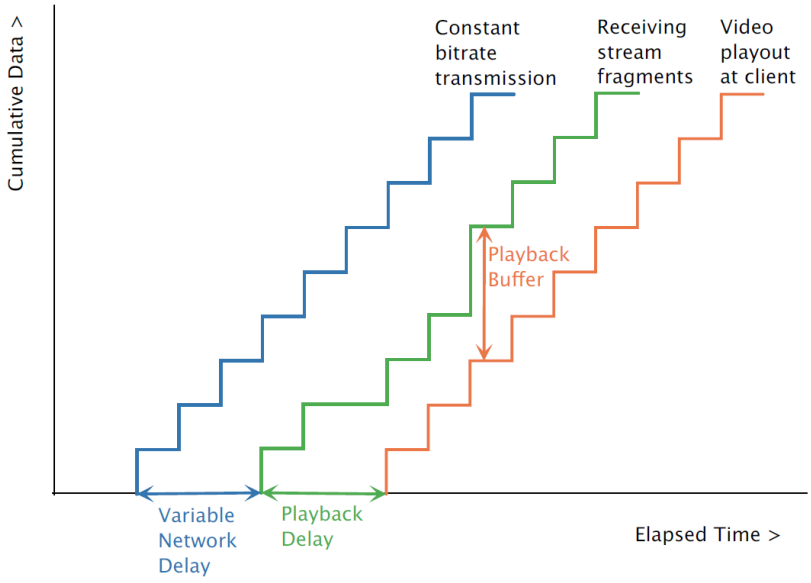

Abbildung 15: On-Demand Streaming (stored video), Quelle: [46]

Die blaue Linie stellt die Übertragung bei konstanter Bitrate dar. Da es im Netzwerk jedoch meist zu Jitter, also zu Variationen in der Netzwerkverzögerung, kommt, kommen die Pakete des Streams ungleichmäßig beim Client an (siehe grüne Linie) [46]. Daher werden die eingehenden Pakete zunächst in einem Puffer gespeichert und die Wiedergabe wird verzögert. Erst nach Ablauf des Playback-Puffers (siehe orangene Linie) werden die Dateien kontinuierlich abgespielt [46].

Im Gegensatz dazu werden die Medien beim Live-Streaming nahezu in Echtzeit aufgenommen, komprimiert und übertragen [46]. Zu den bekanntesten Live-Stream-Plattformen gehören Twitch und YouTube [47]. Aus technischer Perspektive benötigt Live-Streaming deutlich mehr Rechenleistung. Folgende Abbildung zeigt die gesamte übertragene Datenmenge in Abhängigkeit vom zeitlichen Verlauf beim Live-Streaming.

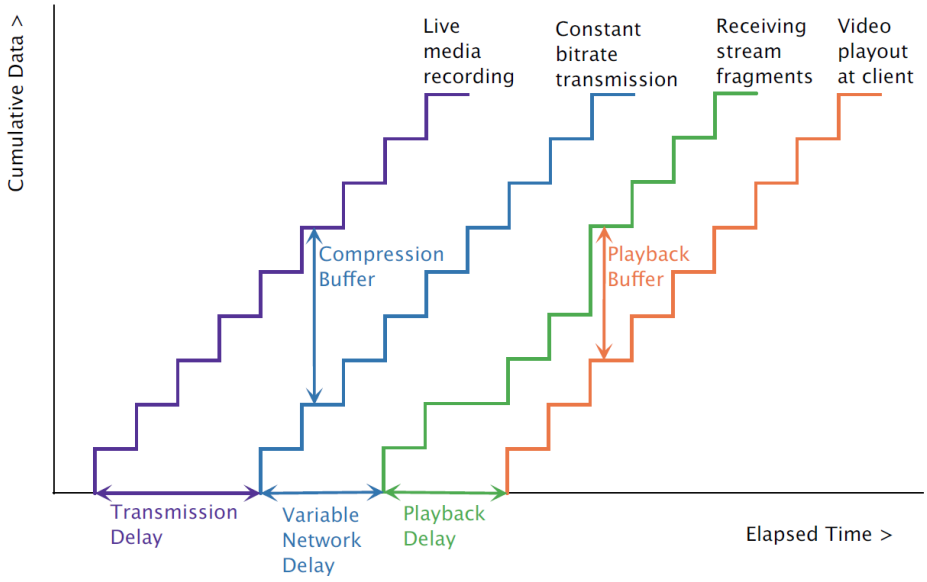

Abbildung 16: Live-Streaming, Quelle: [46]

Das Live-Streaming beginnt mit der eigentlichen Aufnahme der Medien (siehe lila Linie) [46], die im nächsten Schritt komprimiert werden. Es wird also ein zusätzlicher Puffer für die Kompression benötigt, sodass die Übertragung verzögert wird. Erst danach werden die Daten, wie beim On-Demand-Streaming, übertragen und auf der Client-Seite im Playback-Puffer gespeichert. Die Wiedergabe ist beim Live-Streaming nicht nur um den sogenannten ‚Playback Delay' verzögert, sondern auch um die Zeit, die notwendig ist, um die Daten vor der Übertragung zu komprimieren [46].

5.3. Entwicklung des Streamings

Der Begriff ‚Streaming' kam erstmals in den 1970er Jahren zum Einsatz [47]. Die ersten Experimente im Streaming-Bereich begannen um diese Zeit mit der Übertragung von Audioinhalten in den USA. Kurz darauf folgten erste Versuche, Videokonferenzen zu halten [33]. Damals bestand die große Herausforderung in den Netzwerken, die noch nicht genügend Kapazität hatten, um Medieninhalte, außer Text, in Echtzeit über längere Strecken hinweg zu übertragen [47]. Darüber hinaus gab es noch keine Protokolle, die dies ermöglichten [47].

In den 1990er Jahren entwickelte sich schließlich das Audio-Streaming, sodass 1991 das erste Audio-Streaming über eine größere Distanz ermöglicht wurde [47]. Um die Jahrtausendwende begannen zunehmend viele Radiosender damit, ihre Programme auch über das Internet zu übertragen, und es wurden reine Internet-Radiostationen gegründet. Teilweise boten die Sender an, zwischen verschiedenen Programmen zu wählen, was dem gegenwärtigen On-Demand-Streaming ähnelte [47]. Im Jahr 1995 wurde schließlich der erste Streaming-fähige Media-Player, der *RealPlayer*, von RealNetworks eingeführt, der allerdings nur für das Audio-Streaming gedacht war [48].

Das Video-Streaming blieb in den 1990er Jahren eine größere Herausforderung, insbesondere in Bezug auf die großen Datenmengen und Verzögerungszeiten [49]. Nach der Entwicklung des Real-Time Transport Protokolls (RTP), das unter anderem Paketformate für Audio- und Videoinhalte definiert, war es aber möglich, Pakete effizienter zu übertragen [49]. Der erste offizielle Stream mit Audio und Video wurde am 24. Juni 1993 von der Band „Severe Tire Damage“ im privaten Rahmen ausgestrahlt [48].

Mitte der 1990er Jahre waren die verfügbaren Bandbreiten groß genug und die Komprimierungsverfahren und Transportprotokolle weit genug entwickelt, damit das Video-Streaming in einem kleinen Rahmen zum Einsatz kam [33]. Das kommerzielle Live-Video-Streaming war schließlich 1997 möglich, nachdem *RealNetworks* das Programm *RealVideo* einführte [48]. Um diese Zeit kam ebenso der Flash Player von Macromedia, später Adobe, auf den Markt. Von 1996 bis 1998 wurden zudem zwei relevante Streaming-Protokolle entwickelt: RTMP (Real Time Messaging Protocol) und später RTSP (Real Time Streaming Protocol) [3], auf die im weiteren Verlauf dieses Kapitels näher eingegangen wird. Im Jahr 1999 folgte außerdem der Windows Media Player 6 von Microsoft und machte *RealAudio* Konkurrenz [33]. Im selben Jahr kam QuickTime 4.0 von Apple auf den Markt. Dies war schließlich der erste Mediaplayer von Apple, der echte Streaming-Kapazitäten bot [33].

Anfang der 2000er Jahre erfuhr die Streaming-Industrie schließlich großes Wachstum [33]. So wurde im Oktober 2002 das ABR-Streaming eingeführt [3], was für eine zuverlässigere Streaming-Qualität sorgte [47]. Im Jahr 2003 konnte in den USA ein Wachstum von etwa 104 % im Streaming-Bereich verzeichnet werden [33].

Einen besonders großen Durchbruch für das Video-Streaming brachte schließlich die Einführung von YouTube im Jahr 2005 [47]. Drei Jahre später hostete die Plattform ihr erstes Live-Event, begann aber erst in den 2010er Jahren mit der Entwicklung einer eigenen Streaming-Technologie [48].

Der Video-Streaming-Anbieter Netflix begann erstmals im Jahr 2007 mit dem Streaming und der Musik-Streaming-Service Spotify nur etwa ein Jahr danach [47]. Im November 2009 kam die Einführung des sogenannten ‚HTTP-Live-Streaming-(HLS)-Protokolls‘, das mit adaptiver Bitrate arbeitet [3]. Die für Videospiele gedachte Streaming-Plattform Twitch.tv wurde im Jahr 2011 gegründet und erfuhr großen Erfolg mit über 45 Millionen Zuschauern pro Monat nach den ersten zwei Jahren [48]. Seitdem haben sich viele Anbieter auf dem Streaming-Markt etabliert. In den USA allein gab es bereits im Jahr 2015 mehr als 100 verschiedene Streaming-Anbieter [47].

Seit 2013 ist es jedem YouTube-Nutzer möglich, selbst Live-Streams auszustrahlen. Um diese Zeit sind auch andere Social-Media-Plattformen in das Streaming-Geschäft eingestiegen. Seitdem wachsen die Nutzerzahlen stetig [48]. Im Jahr 2020 wurde YouTube schließlich der größte Video-Streaming-Service [47].

Bereits 2016 wurde der Streaming-Markt zur wichtigsten Einnahmequelle weltweit [47], jedoch stiegen die Nutzerzahlen im Streaming-Geschäft insbesondere während der Corona-Pandemie noch weiter an.

So nahm die Anzahl der Live-Streaming-Events von Mai bis August 2020 um 300 % zu [3]. Es wird angenommen, dass die Live-Streaming-Industrie auch in der Zukunft weiterhin wachsen wird [48].

Die sich über Jahrzehnte entwickelte Streaming-Technologie spielt jedoch nicht nur für die Unterhaltungsindustrie eine wichtige Rolle. Sie wird auch eingesetzt, um zum Beispiel Live-Videosignale von Sicherheitskameras zu übertragen oder wie in diesem Projekt den Bildschirminhalt entfernter Rechner als Stream in einer Webapplikation bereitzustellen.

5.4. Streaming-Protokolle

Streaming-Protokolle sind Transportprotokolle, die dafür zuständig sind, Mediendaten über das Internet zu übertragen [50]. Sie definieren Regeln, die bestimmen, wie die Daten über das Netzwerk übertragen werden, und sind in verschiedene Komponenten unterteilt, wie die Header, die eigentlichen Daten, die Authentisierung und das Error Handling [50]. Zwei Transportprotokolle, die zur Datenübertragung eingesetzt werden, wurden bereits vorgestellt (siehe 4.3.). Nun soll ein Blick auf Streaming-Protokolle geworfen werden.

Streaming-Protokolle werden allgemein in sogenannte ‚Push- und Pull-basierte' Protokolle unterschieden [50]. Push-basierte Streaming-Protokolle basieren meist auf dem *User Datagram Protocol* und nutzen in der Regel das *Real-Time-Transport*-Protokoll (RTP). Nachdem eine Verbindung zwischen Client und Server aufgebaut wurde, sendet der Server so lange einen Stream, bis der Client die Kommunikation unterbricht oder beendet [50]. Im Gegensatz dazu basieren Pull-basierte Streaming-Protokolle auf dem HTTP-Protokoll [50]. Der Client ist dafür verantwortlich, eine Anfrage an den Server zu senden. Die Kommunikation läuft so ab, dass der Client die Streaming-Dateien herunterlädt. Bei diesen Protokollen ist die Streaming-Geschwindigkeit von der Bandbreite des Netzwerkes abhängig, in dem sich der Client befindet [50]. Folgende Übersichtstabelle zeigt verschiedene Streaming-Protokolle im Vergleich, die in den folgenden Abschnitten näher vorgestellt werden.

	RTP	RTMP	HLS	MPEG-DASH	WebRTC	QUIC
Video-Codec	JPEG, H.261, H.263 [51] H.264 [52]	H.264 [53]	H.264 / H.265 [46]	alle [54]	VP8, VP9, H.264 [53]	Abhängig von Streaming-Protokoll
Audio-Codec	iLBC, vorbis, etc. [55]	AAC [53]	AAC, MP3 [46]	alle [54]	opus, iSAC, iLBC [56]	Abhängig von Streaming-Protokoll
Latenz	< 2s [57]	3–30s [53]	Standard: 6–30s Low-Latency: ca. 2s [56]	Standard: 6–30s [58] Low-Latency: 4–5s [59]	< 1s [53]	u. a. abhängig von Streaming-Protokoll
ABR	nein	nein [60]	ja [45]	ja [54]	ja [53]	ja
Transport-protokoll	UDP [51]	UDP oder TCP [61]	TCP [62]	TCP [63]	UDP [64]	UDP [65]
Anmerkung	Flash veraltet →nur noch als ingest genutzt [40]	Flash veraltet →nur noch als ingest genutzt [40]		Open-Source-Alternative zu HLS [54]	Protokoll-Sammlung für P2P-Kommunikation [64]	universelles Transport-protokoll [66]

Tabelle 2: Übersicht Streaming-Protokolle, eigene Darstellung in Anlehnung an [53]

5.4.1. RTP + RTCP

Das *Real-Time-Transport*-Protokoll (RTP) ist ein Transportprotokoll, das auf UDP aufgebaut und Teil der Anwendungsschicht ist [51]. Es ist eine Alternative zu den reinen Transportprotokollen TCP und UDP, um Echtzeitkommunikation zu ermöglichen. Wie bereits erwähnt wurde, geht TCP meist aufgrund diverser Kontrollmechanismen mit längeren Verzögerungszeiten einher. Dahingegen ermöglicht UDP zwar kürzere Verzögerungszeiten, aber auch eine unzuverlässige Übertragung, da kein Mechanismus zum Erhalt der Paketreihenfolge vorhanden ist [67]. RTP wurde im Jahr 1996 veröffentlicht [68], also vor der Einführung des ABR-Streamings, veröffentlicht, um Audio- und Videoinhalte ohne die Nachteile von TCP und UDP über das Internet zu übertragen.

Beim RTP-Streaming werden sogenannte ‚Chunks', also Segmente, aus eingehenden Video- oder Audiosignalen generiert und in RTP-Pakete verkapselt. Da RTP auf UDP basiert, werden die RTP-Pakete wiederum als UDP-Segmente verkapselt und versendet [69]. RTP bietet dabei Zeitstempel und Sequenznummern, über die UDP nicht verfügt. Dies stellt sicher, dass Pakete beim Empfänger in richtiger Reihenfolge wiedergegeben werden können [69]. Zudem kann mithilfe der Sequenznummern festgestellt werden, wie viele Pakete verloren gehen [69].

Das RTP Control Protocol (RTCP) ist ein Kontrollprotokoll, das in Kombination mit RTP genutzt wird [51]. Es dient dazu, zum Beispiel Teilnehmern einer Video- oder Audiokonferenz Rückmeldung bezüglich der Übertragungsqualität und Informationen über die Streaming-Teilnehmer zu liefern [51]. RTCP-Pakete werden dabei in regelmäßigen Abständen von Streaming-Teilnehmern an alle anderen Teilnehmer übertragen.

RTP unterstützt eine Vielzahl verschiedener Video- und Audio-Codecs, darunter zum Beispiel H.263, H.264, vorbis und opus [52]. Dieses Streaming-Protokoll erreicht dabei Latenzen von unter zwei Sekunden [57]. Allerdings wurde es insbesondere für die Übertragung von Streams an Flash-Videoplayer eingesetzt, die mittlerweile veraltet und von den meisten Browsern nicht mehr unterstützt werden. Daher wird RTP gegenwärtig nur noch für die Übertragung vom Encoder zum Streaming-Server eingesetzt. Für die Übertragung vom Server an die Clients wird häufig auf neuere Protokolle zurückgegriffen.

5.4.2. RTMP + RTSP

Das *Real-Time-Messaging*-Protokoll (RTMP) ist ein Netzwerkprotokoll, das Audio- und Videodateien zwischen Flash-Plattformen überträgt [70]. Ursprünglich von Macromedia (heute Adobe) entwickelt, wurde es anfangs vor allem dazu verwendet, Inhalte zwischen Streaming-Servern und Flash-Videoplayern zu übertragen [61]. Da Flash veraltet ist, wird RTMP gegenwärtig nicht mehr dazu genutzt, die eigentlichen Inhalte an Videoplayer zu übertragen, sondern lediglich dafür, die Inhalte vom Encoder an den Streaming-Server beziehungsweise Online-Video-Host zu senden [61], was als ‚*ingestion*' bezeichnet wird (siehe 5.2.).

Bei einem *RTMP ingest* wird die RTMP-Technologie dafür verwendet, ein Eingangssignal, wie einen Audio- oder Videostream, zu kodieren und anschließend an eine Live-Video-Plattform zu liefern [40]. Ein *RTMP ingest* besteht aus drei Phasen: dem Handshake (1), dem Connect (2) und dem Stream (3) [40]. Während des Handshakes werden drei Pakete vom Client an den Server übertragen. Als erstes wird eine Benachrichtigung an den Server geschickt, die diesem mitteilt, welches Protokoll genutzt wird. Das nächste Segment liefert einen Zeitstempel und das dritte Paket dient der Bestätigung [40]. In der Connect-Phase tauschen Client und Server verschiedene, kodierte Nachrichten aus, um eine Verbindung aufzubauen. Sobald die Verbindung aufgebaut ist, kann in der letzten Phase, dem Stream, mit der Übertragung des Streams begonnen werden [40].

Da ein *RTMP-ingest* lediglich die Übertragung an den Live-Streaming-Server übernimmt, wird der Stream häufig per HLS (siehe 5.4.3.) an den Videoplayer übertragen [40].

Das Real-Time-Streaming-Protokoll (RTSP) wird eingesetzt, um die Übertragung von Echtzeitdaten über ein Netzwerk zu kontrollieren [71]. Dabei ist es nicht für die eigentliche Datenübertragung zuständig, sondern wird in Kombination mit RTP oder RTMP genutzt, um die Übertragung zu steuern [71]. Es handelt sich um ein verbindungsloses Protokoll, das sich auf der Anwendungsschicht befindet [50].

Das RTSP-Protokoll unterstützt drei verschiedene Operationen. Zum einen wird es dazu genutzt, Medieninhalte von einem Server abzurufen [50]. Andererseits kann RTSP eingesetzt werden, um Medien-Server zu Konferenzen beizufügen, um zum Beispiel Mediendateien abzuspielen oder diese

aufzunehmen [50]. Letztendlich findet das Protokoll Einsatz, wenn es darum geht, Clients über neue Mediendaten zu informieren [50].

RTSP-URLs ähneln denen des HTTP-Protokolls und sind nach dem Schema *rtsp://* aufgebaut [50]. Folgende Abbildung zeigt die verschiedenen Methoden, die vom RTSP-Protokoll genutzt werden, um einen Stream zu empfangen.

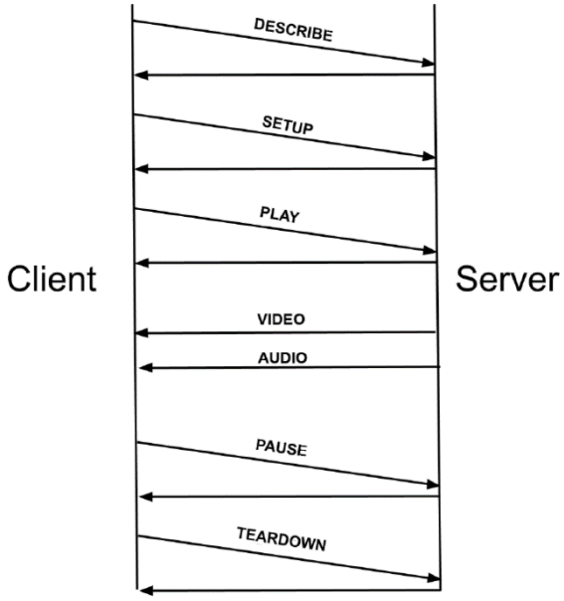

Abbildung 17: RTSP Request-Reihenfolge, Quelle: [50]

Bevor das Streaming beginnen kann, wird im ersten Schritt eine TCP-Verbindung zwischen dem Client und dem Server aufgebaut, die vom Client ausgeht. Sobald die TCP-Verbindung zwischen Client und Server erfolgreich aufgebaut wurde, kann der Client die Methode DESCRIBE nutzen, um in der Initialisierungsphase eine Beschreibung der angeforderten Ressource zu erhalten. Dies wird mithilfe des *Session Description Protocols* realisiert [50]. Der Server schickt als Antwort eine Liste der angeforderten Multimedia-Streams [50].

Um im nächsten Schritt zu definieren, wie der Stream übertragen wird, schickt der Client eine Anfrage mit der URL des gewünschten Multimedia-Streams und die Spezifikationen für die Übertragung, wie zum Beispiel den Port, auf dem der RTSP-Stream empfangen werden soll [50]. Dazu wird die Methode SETUP verwendet [50]. Der Server antwortet mit einer Bestätigung [50].

Schließlich kann eine PLAY-Anfrage an den Server geschickt werden, um das Streaming zu starten [50]. Die PAUSE-Methode pausiert die Übertragung kurzfristig und ein TEARDOWN-Request beendet die Übertragung des Streams [50].

RTMP unterstützt wie RTP eine Vielzahl verschiedener Audio- und Video-Codecs, darunter zum Beispiel H.264 und AAC [53]. Die Latenz liegt beim Streaming zwischen drei und 30 Sekunden [53]. RTMP bietet kein adaptives Bitrate-Streaming [60] und basiert je nach Variante auf TCP oder auf UDP

[61]. Auch dieses Protokoll wird hauptsächlich als *ingest* verwendet, da Flash nicht mehr unterstützt wird.

5.4.3. HLS

Das *HTTP-Live-Streaming*-Protokoll (HLS) wurde 2009 von Apple entwickelt [54]. Ursprünglich war es dazu gedacht, das Live-Streaming auf dem iPhone zu ermöglichen, jedoch unterstützen gegenwärtig fast alle Geräte das HLS-Streaming [54]. Bei diesem handelt es sich um ein zuverlässiges Streaming-Protokoll, das auf HTTP basiert und es ermöglicht, Videos kontinuierlich über das Internet zu übertragen [72]. Es arbeitet mit ABR, sodass Empfänger die Bitrate des Streams dynamisch an die Netzwerkbedingungen anpassen können [45]. Folgende Abbildung zeigt die Architektur des HLS.

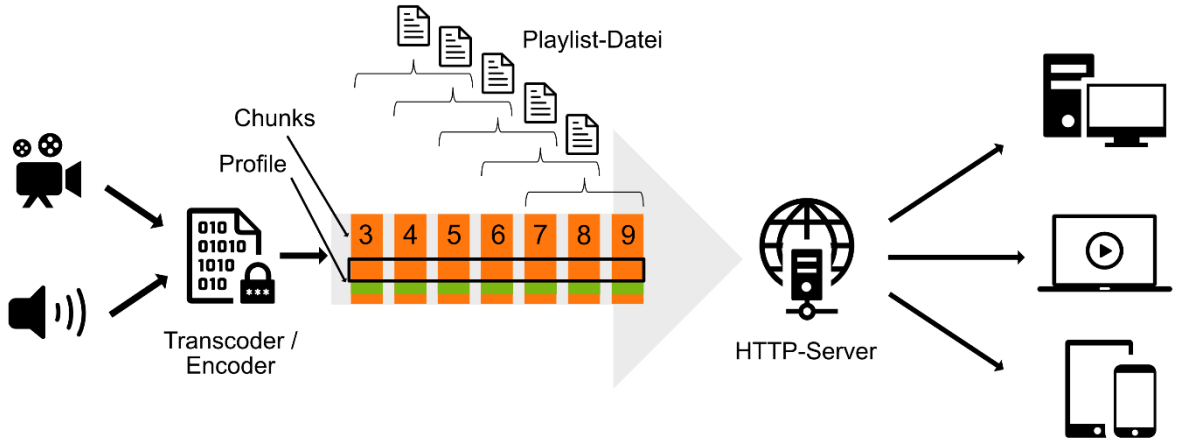

Abbildung 18: HLS-Architektur, eigene Darstellung in Anlehnung an [73]

In einem ersten Schritt wird der Input-Stream, zum Beispiel ein Video- oder Audio-Stream, mithilfe eines Encoders in einen MPEG-2 Transport Stream Container kodiert [45]. Daraufhin wird der kodierte Input-Stream, wie bei allen Protokollen, die auf ABR basieren, in verschiedene Profile mit unterschiedlich hoher Auflösung beziehungsweise Bitrate kodiert [45].

Danach werden die unterschiedlich kodierten Streams in einzelne Mediensegmente, sogenannte ‚*Chunks*', unterteilt, die beim kontinuierlichen Abspielen den Stream ergeben [45]. Während der Segmentierung werden gleichzeitig auch ‚Media-Playlists' und eine ‚Master-Playlist' angelegt. Media-Playlists enthalten Informationen zu den verfügbaren Chunks, wie zum Beispiel die jeweilige Länge eines Segments und die URI, unter der ein Chunk heruntergeladen werden kann [72]. Dahingegen ist eine Master-Playlist eine komplexere Beschreibung der Chunks und enthält zusätzliche Chunk-Informationen, wie zum Beispiel die verfügbaren Bitraten, Medienformate und Auflösungen, wenn ein Stream in unterschiedlichen Profilen repräsentiert ist [72].

Möchte ein Client einen Stream abrufen, lädt dieser zunächst die entsprechende Playlist herunter [45]. Der Client kann dann zwischen den verschiedenen Bitraten und Auflösungen wählen und die einzelnen

Segmente mit Hilfe der in der Playlist enthaltenen URLs herunterladen [45]. Chunks sind in der Playlist mit Sequenznummern gekennzeichnet, sodass diese reihenfolgegetreu abgerufen werden können. Sobald der Client eine minimale Anzahl an Chunks heruntergeladen und gepuffert hat, beginnt die Wiedergabe des Streams, indem die Chunks hintereinander abgespielt werden [45].

Dementsprechend entspricht die minimale Latenz in Live-Streaming-Anwendungen mindestens der Segment-Länge [74]. Beim Puffern einer bestimmten Anzahl an Chunks wird die Latenz abhängig von der Konfiguration um zwei bis drei Segment-Längen erhöht. Durchschnittlich beträgt die Latenz dabei mehrere zehn Sekunden, was für Live-Streaming-Anwendungen ungeeignet ist [74]. Eine naheliegende Lösung für die Reduktion der Latenz ist das Verringern der Segment-Länge. Allerdings muss beachtet werden, dass damit die Anzahl an Server-Anfragen und damit die Netzlast steigen, da jedes Segment einzeln vom Server angefordert wird [74].

Die durchschnittlich erreichbare Latenz beim Streaming mit HLS liegt bei sechs bis 30 Sekunden. Allerdings wurde eine Low-Latency-Variante entwickelt, die Latenzen von etwa zwei Sekunden ermöglicht [56]. Um diese zu erreichen, kann, wie zuvor beschrieben wurde, die Segmentlänge reduziert werden. Da HLS auf TCP basiert, ist generell mit höheren Latenzen als bei RTP und RTMP zu rechnen. Allerdings bietet HLS dadurch eine zuverlässige Übertragung der Streaming-Segmente.

5.4.4. MPEG-DASH

Das *Dynamic Adaptive Streaming over HTTP* (MPEG-DASH) ist ein Streaming-Protokoll, das als open-source-Alternative zum von Apple entwickelten HLS im Jahr 2012 standardisiert wurde. Es bietet auch eine Form des adaptiven Bitrate-Streamings und beruht auf dem HTTP-Protokoll [49]. Folgende Abbildung zeigt den generellen Ablauf des Streamings mit MPEG-DASH.

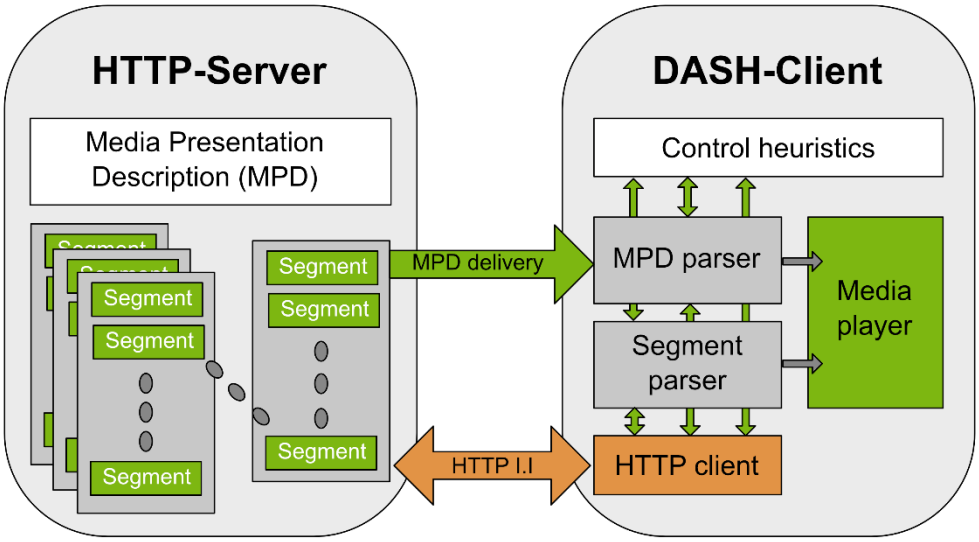

Abbildung 19: MPEG-DASH-Architektur, eigene Darstellung in Anlehnung an [49]

Beim Streaming mit MPEG-DASH werden die Medieninhalte auf einem HTTP-Server in Form von Segmenten abgelegt [49]. Segmente enthalten die Bitstreams der Inhalte, wobei meist mehrere Segmente denselben Inhalt in unterschiedlichen Kodierungen enthalten. Neben den eigentlichen Segmenten ist außerdem eine sogenannte ‚*Media Presentation Description*' (MPD) auf dem Server abgelegt. Diese ist eine Art Manifest-Datei, vergleichbar mit der Master-Playlist beim HLS, und enthält Informationen über die verfügbaren Kodierungen, dessen URLs und weitere Eigenschaften der Segmente [49].

Um die Segmente abzurufen und einen Stream abzuspielen, fordert ein DASH-Client im ersten Schritt die MPD an [49]. Ein MPD-Parser verarbeitet die Datei, sodass der Client Auskunft über die zur Verfügung stehenden Inhalte erhält. Dazu zählen die verfügbaren Kodierungen, Auflösungen und Medientypen sowie maximale Bandbreiten und weitere Eigenschaften der Inhalte [49].

Mittels HTTP GET-Request fordert der Client nun die passenden Segmente vom Server an und puffert sie für eine bestimmte Zeit [49]. Dabei kontrolliert der DASH-Client regelmäßig die Netzwerkbandbreite, die von Zeit zu Zeit variieren kann. Abhängig von der verfügbaren Bandbreite reagiert der Client, indem er alternative Segmente mit niedrigerer oder höherer Bitrate anfordert [49]. Dies ermöglicht Nutzern, Streams in Abhängigkeit von ihrer aktuellen Internetgeschwindigkeit in der jeweils bestmöglichen Qualität zu empfangen.

Spezifiziert durch MPEG-DASH sind einerseits die MPD und andererseits das Format der Segmente [49]. Die Übertragung sowie die Kodierungsformate sind nicht im MPEG-DASH-Standard definiert, sodass verschiedene Formate eingesetzt werden können [54].

MPEG-DASH bietet standardmäßig Latenzen von etwa sechs bis 30 Sekunden an [58]. Allerdings können beim Einsatz des sogenannten ‚*Common Media Application Format*' (CMAF) auch niedrigere Latenzen von vier bis fünf Sekunden erreicht werden [59]. Wie HLS basiert dieses Streaming-Protokoll auf TCP, was für eine zuverlässige Übertragung sorgt.

5.4.5. WebRTC

Web Real-Time Communication (WebRTC) ist eine Protokoll-Sammlung, die im Jahr 2011 von Google entwickelt wurde, um Echtzeitkommunikation im Internet, wie zum Beispiel Online-Audio- und Videochats, zu ermöglichen [75]. Im Gegensatz zur herkömmlichen Browser-Server-Kommunikation ermöglicht WebRTC Browsern, direkt miteinander zu kommunizieren, sodass es sich hierbei um Peer-to-Peer-Kommunikation handelt [64]. Dabei funktioniert dies ohne Plugins oder zusätzliche Software von Drittanbietern. Mittlerweile hat das *World Wide Web Consortium* (W3C) eine API (*Application Programming Interface*) entwickelt, die es Entwicklern ermöglicht, die WebRTC-Technologie in eigene Anwendungen zu integrieren. Die Internet Engineering Task Force (IETF) hat die entsprechenden Protokolle und Formate definiert [64]. Folgende Abbildung zeigt die Architektur der WebRTC-Technologie.

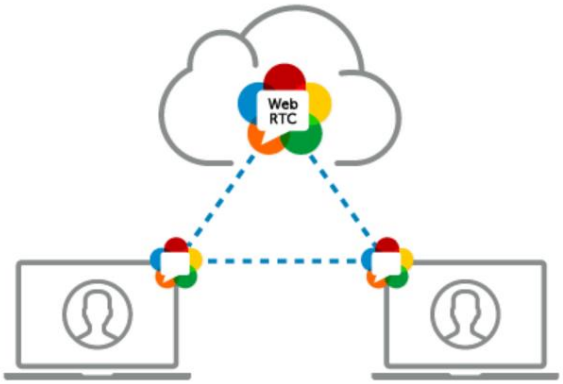

Abbildung 20: WebRTC-Architektur, Quelle: [56]

WebRTC basiert auf sogenannten ‚*RTCPeerConnections*', um Streaming-Daten zwischen Browsern auszutauschen, ohne dass ein Streaming-Server benötigt wird [76]. Dennoch ist ein Mechanismus notwendig, um die Kommunikation zu koordinieren [76], was auch als ‚*Signaling*' bezeichnet wird [76]. Dabei ist ein Signaling-Server notwendig, um Kontrollnachrichten auszutauschen, zum Beispiel eine Kommunikation aufzubauen oder zu schließen, das Netzwerk zu konfigurieren und zu bestimmen, welche Codecs oder Auflösungen der entsprechende Browser unterstützt [76].

Die drei Hauptkomponenten der API sind *MediaStream*, *RTCPeerConnection* und *RTCDataChannel* [75]. *MediaStream* ermöglicht dem Browser, auf die Kamera und das Mikrofon zuzugreifen [75]. *RTCPeerConnection* dient dazu, Audio- oder Videoanrufe aufzubauen, und *RTCDataChannel* bietet Browsern die Möglichkeit, Daten über Peer-to-Peer-Verbindungen auszutauschen [75].

Für das Audio- oder Videostreaming können, ähnlich wie bei anderen Streaming-Protokollen, Codecs wie H.264 oder opus eingesetzt werden. Allerdings bietet WebRTC im Vergleich zu anderen Streaming-Protokollen die derzeit geringsten Verzögerungszeiten [56]. Die Latenz liegt dabei unter einer Sekunde, wodurch sich die Protokoll-Sammlung gut für Echtzeitanwendungen wie Videokonferenzen eignet [56].

5.4.6. QUIC

Das *Quick UDP Internet Connections* Protokoll (QUIC) ist ein Transportprotokoll, das auf UDP basiert und von Google entwickelt wurde, um HTTP über TCP zu ersetzen und die Latenzen bei der Datenübertragung über das Internet zu reduzieren [65]. Wie in Kapitel 4.4.1. thematisiert wurde, kann es beim Transportprotokoll TCP, das unter anderem für HLS und MPEG-DASH zum Einsatz kommt, aufgrund diverser Kontrollmechanismen wie der Staukontrolle zu größeren Verzögerungen bei der Datenübertragung kommen. Insbesondere beim Streaming sind solche Latenzen jedoch zu vermeiden, weshalb QUIC auf UDP aufgebaut ist, mit dem Ziel, lange Verzögerungszeiten zu vermeiden [77].

Da UDP allerdings verbindungslos und unzuverlässig ist, implementiert QUIC eine Staukontrolle und überträgt verlorene Pakete erneut [77]. Unterschiede von QUIC im Vergleich zu TCP sind vor allem der

Verbindungsaufbau, der in folgender Abbildung dargestellt ist, und das Multiplexing, auf das im Verlauf dieses Abschnitts eingegangen wird.

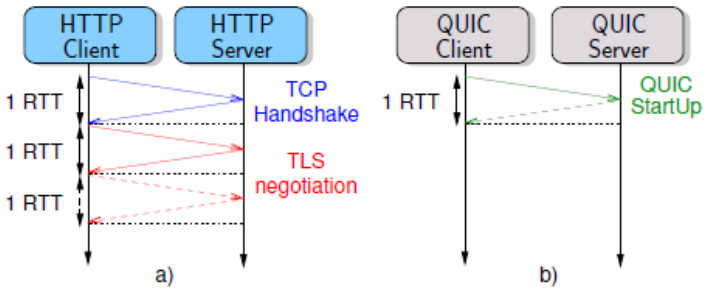

Abbildung 21: HTTP vs. QUIC (Verbindungsaufbau), Quelle: [77]

Abbildung 21 zeigt die Zeit, die für den Verbindungsaufbau bei TCP und QUIC benötigt wird, bevor die eigentlichen Daten übertragen werden können. TCP nutzt für den initialen Verbindungsaufbau zwischen Client und Server einen 3-Wege-Handshake (siehe 4.4.1.). Aufgrund des 3-Wege-Handshakes und der Verhandlung verschiedener Informationen benötigt TCP mindestens zwei RTT (*Round-Trip-Time*). Wenn eine verschlüsselte Verbindung aufgebaut wird, liegt die Zeit für den Verbindungsaufbau sogar bei drei RTT [77]. Im Vergleich dazu benötigt QUIC höchstens eine RTT für den initialen Verbindungsaufbau. Wenn der Client und der Server erneut eine Verbindung aufbauen, also schon einmal verbunden waren, ermöglicht QUIC den direkten Datenaustausch ohne weitere RTT, was die Latenz im Vergleich zu TCP stark reduziert.

Ein weiterer Unterschied zwischen QUIC und TCP ist die Art und Weise, wie Datenströme übertragen werden. Während bei TCP lediglich ein Datenstrom pro TCP-Verbindung übertragen wird, ermöglicht QUIC es, mehrere Datenströme unabhängig voneinander über dieselbe Verbindung zu übertragen (siehe Abbildung 22).

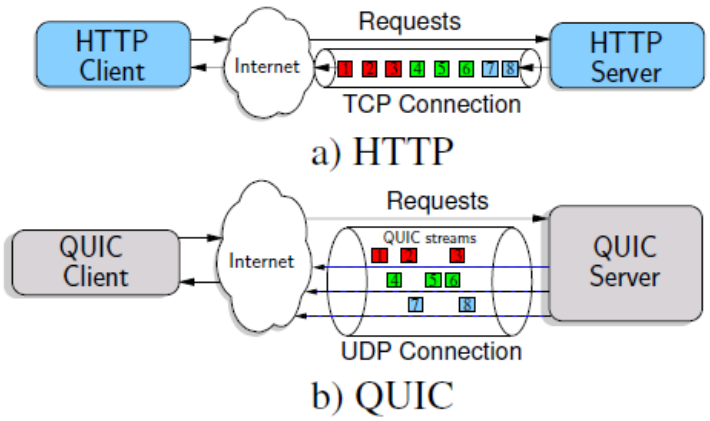

Abbildung 22: HTTP vs. QUIC (Multiplexing), Quelle: [77]

TCP sowie QUIC nutzen Sequenznummern, um es dem Empfänger zu ermöglichen, eingehende Daten in die ursprüngliche Reihenfolge zu sortieren. Das Problem bei TCP besteht jedoch darin, dass der Empfänger bei Paketverlust im Datenstrom aufhört, die darauffolgenden Pakete zu verarbeiten [78]. Der

Empfänger wartet dabei so lange, bis das verlorene Paket wiederholt übertragen wurde und eintrifft. Dies wird auch als ‚*head of line blocking*' (HOL) bezeichnet [78].

Der Vorteil bei QUIC besteht darin, dass die Auswirkungen des HOL durch das Multiplexing reduziert werden. Das bedeutet, dass bei Paketverlust lediglich der Stream blockiert wird, in dem Paketverlust aufgetaucht ist [79]. Die anderen Streams können unabhängig davon vom Empfänger verarbeitet werden.

QUIC ist ein universelles Transportprotokoll [66], das unter anderem für Streaming-Anwendungen eingesetzt werden kann. Es bietet adaptives Bitrate-Streaming und wird bereits von Google und YouTube eingesetzt. Derzeit werden etwa 7 % des Internetverkehrs über QUIC übertragen [65].

6. Marktübersicht

In diesem Kapitel wird eine Marktübersicht über die wichtigsten Streaming-Komponenten präsentiert. Dazu zählen Encoder und unterstützte Codecs, Streaming- beziehungsweise Medien-Server sowie Video-Player. Dieses Kapitel dient als Grundlage für die im nächsten Kapitel behandelten Lösungsansätze, die von den genannten Streaming-Komponenten Gebrauch machen.

6.1. Encoder

Encoder werden dazu verwendet, eingehende, unkomprimierte Signale wie Audio und Video in ein komprimiertes Format zu konvertieren, sodass sie effizient über das Internet-Netzwerk übertragen werden können [38]. Encoder können hardware-, aber auch softwarebasiert sein. Außerdem gibt es spezielle Live-Streaming-Encoder, die Videosignale in Echtzeit verarbeiten können [80].

Hardware-Encoder sind eigenständige Geräte, die unkomprimierte Video- oder Audiosignale, meist über HDMI- oder SDI-Eingänge, empfangen, in ein komprimiertes, standardisiertes Format konvertieren und schließlich mittels eines standardisierten Protokolls über IP übertragen [38]. Dahingegen laufen Software-Encoder auf einem Rechner, auf dem meist auch andere Prozesse laufen, sodass es sich beim Encoding nicht um die Hauptaufgabe des Rechners handelt [80].

Generell sind Hardware-Encoder im Vergleich zu Software-Encodern effizienter, da sie rein für das Encoding ausgelegt sind [80]. Allerdings sind sie teurer als Software-Encoder, sodass sie eher im professionellen Bereich Anwendung finden. Für Einstiegsprojekte oder Anwendungen im mittleren Bereich werden häufig Software-Encoder eingesetzt. Teilweise sind sie sogar frei verfügbar und meist einfacher zu bedienen [80].

Ziel dieses Projektes ist es, eine Streaming-Lösung zu finden, die auf einem Server in der Cloud läuft, weshalb für dieses Projekt lediglich Software-Encoder in Frage kommen. Außerdem soll es sich um einen Encoder handeln, der für das Live-Streaming optimiert ist und Videosignale in Echtzeit kodieren kann. Zudem wäre es optimal, wenn es sich um eine frei verfügbare Software handelt. Dennoch werden zur Vollständigkeit der Marktübersicht im Folgenden auch einige proprietäre Encoder vorgestellt.

6.1.1. Open Broadcaster Software (OBS)

Open Broadcaster Software (OBS) [81] ist eine open-source-Software, die für diverse Video-Streaming-Anwendungen genutzt werden kann. Dabei handelt es sich genaugenommen nicht nur um einen Software-Encoder, sondern um eine Live-Video-Software. Das bedeutet, dass eingehende Video- und Audiosignale nicht nur kodiert und an einen Streaming-Server geschickt werden können, sondern auch Aufnahmen gemacht und lokal auf einem Rechner gespeichert werden können [81]. Außerdem können Audiosignale gemischt und ganze Szenen aus verschiedenen Eingangssignalen kreiert werden [81]. Dabei ist es auch möglich, Bild- und Textelemente sowie Bildschirm- beziehungsweise Fensteraufnahmen hinzuzufügen [81].

OBS ermöglicht es, Live-Video-Streams über verschiedene Plattformen wie zum Beispiel Twitch, YouTube oder einen eigenen Media-Server zu streamen [82]. Dazu kann in den Streaming-Einstellungen die gewünschte Plattform aus einer Drop-Down-Liste gewählt werden. Außerdem müssen die Adresse und der Streamschlüssel des gewählten Streaming-Servers angegeben werden [83]. Wird zum Beispiel YouTube als Plattform gewählt, sind die Serveradresse und der Streamschlüssel von YouTube vorgegeben. Soll der Stream aber an einen eigenen Streaming-Server gesendet werden, wird die IP-Adresse dieses Servers genutzt und ein eigener Streamschlüssel erstellt [83].

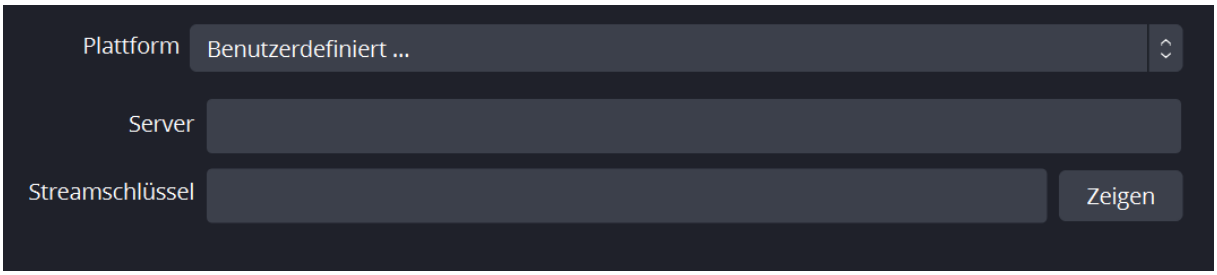

Abbildung 23: OBS-Plattformauswahl, eigener Screenshot

Neben der gewünschten Plattform lassen sich in OBS verschiedene andere Streaming-Einstellungen vornehmen. So können zum Beispiel der genutzte Encoder, die Bitrate oder auch die Qualitätsregulierungsmethode, also ob zum Beispiel ABR eingesetzt wird, definiert werden. Im Durchschnitt lässt sich mit OBS eine Latenz von etwa fünf Sekunden realisieren. Sind die Einstellungen optimiert und ist der ‚low-latency'-Modus ausgewählt, lassen sich aber auch Latenzen von einer bis zwei Sekunden erreichen [82]. Allerdings kann diese Latenz zum Endnutzer nur erreicht werden, wenn der Stream auf der Plattform wiedergegeben wird. Wird der Stream erst an einen Streaming-Server geschickt, eventuell transkodiert und erst dann an den Client geschickt, muss mit zusätzlicher Latenz gerechnet werden.

6.1.2. ffmpeg

ffmpeg ist ein open-source-Video- und -Audio-Konverter, der ein bestimmtes Eingangssignal in ein Ausgangssignal konvertiert [84]. Es handelt sich dabei um ein Kommandozeilen-Tool, das es ermöglicht, Signale oder Dateien zu kodieren, zu transkodieren, zu dekodieren, zu filtern oder auch an einen Medien-Server zu streamen [85]. Es bietet diverse Einstellungen, um zum Beispiel das Format, die Qualität, die Framerate und das Containerformat zu definieren [84].

So ist es möglich, mittels *ffmpeg* einen Videostream an einen RTMP-Server zu übertragen oder aber den Videostream in streamingfähige Chunks für HLS (siehe 5.4.3) aufzuteilen.

Um *ffmpeg* zu nutzen, muss das Tool lediglich auf dem entsprechenden Rechner heruntergeladen werden. Danach können Kommandos in folgendem Format abgesetzt werden, um ein Eingangssignal in ein bestimmtes Ausgangssignal zu konvertieren:

```
ffmpeg -i input-stream options output-stream
```

Abbildung 24: ffmpeg-Kommando, eigener Screenshot

input-stream steht dabei für das Eingangssignal. Es wird immer hinter der Option *-i* als Input definiert und kann eine Audio- oder Videodatei in einem bestimmten Format sein, oder aber ein Stream, wie zum Beispiel ein RTMP-Stream. *options* ist ein Platzhalter für diverse Einstellungen, die vorgenommen werden können, wie zum Beispiel *-r 24* für eine Framerate von 24 Frames pro Sekunde (fps) [84]. Weitere Optionen, die *ffmpeg* bietet, werden im Verlauf dieser Thesis detaillierter erläutert. Als *output-stream* wird in Abbildung 24 das Ausgangssignal definiert, das eine Audio- oder Videodatei in einem bestimmten Ausgangsformat oder ein Stream sein kann.

Um eine einfache Konvertierung eines Flash-Videos in ein MP4-Video durchzuführen, kann folgendes Kommando in der Kommandozeile abgesetzt werden:

```
ffmpeg -i input_file.flv output_file.mp4
```

Abbildung 25: ffmpeg-Kommando, Konvertierung .flv nach .mp4, eigener Screenshot

6.1.3. Proprietäre Encoder

Neben OBS und *ffmpeg* gibt es eine Vielzahl kostenpflichtiger Software-Encoder. Diese bieten meist unterschiedliche Abonnements an, die einen unterschiedlich großen Umfang an Funktionen bieten [86]. Häufig enthalten sie nicht nur die Encoder-Software, sondern wie OBS Möglichkeiten zur Video-Produktion, um zum Beispiel mehrere Videosignale zu kombinieren, Übergänge einzublenden oder Audiosignale zu mischen [87]. Zu diesen Encodern gehören zum Beispiel vMix, Wirecast, VidBlasterX oder XSplit [87]. Da diese jedoch für das Projekt nicht in Frage kommen, werden sie an dieser Stelle nicht weiter vorgestellt.

6.1.4. Integrierte Encoder

Für das (Live-)Streaming werden häufig auch Apps, wie Facebook, eingesetzt, wobei es möglich ist, direkt vom Smartphone aus Inhalte zu streamen. Da die Apps bereits über integrierte Encoder verfügen, ist es nicht notwendig, zusätzliche Encoding-Software zu installieren, um diese Funktionalität nutzen zu können [87].

Eine weitere Möglichkeit ist es, Cloud-basierte Encoding-Software, wie zum Beispiel Restream Studio, zu nutzen [87]. Auch diese Software verfügt über integrierte Encoder, sodass keine zusätzliche Software heruntergeladen werden muss. Solche Lösungen ermöglichen es ebenfalls meist, einen Stream parallel auf mehreren Plattformen zur Verfügung zu stellen [87].

6.1.5. Hardwarebasierte Encoder

Hardwarebasierte Encoder sind eigenständige Geräte, die sich speziell um die Kodierung von Eingangssignalen kümmern. Sie sind dementsprechend schneller und zuverlässiger, aber dafür auch teurer und weniger individualisierbar [86]. Einige bekannte Hardware-Encoder sind zum Beispiel *LiveU Solo*, *Teradek* oder *TriCaster*.

LiveU Solo ist ein weltweit bekannter Encoder, der bereits für Weltmeisterschaften und die Olympischen Spiele eingesetzt wurde [80]. Er ist demnach für Live-Streaming geeignet und unterstützt Remote-Streaming [80]. *Teradek* ist eine weitere Option, die für das Streaming mit geringer Verzögerung geeignet ist. Außerdem ermöglicht sie das mobile Streaming und kann abhängig von der Variante auch unterwegs eingesetzt werden [80]. *TriCaster* ist ein Echtzeit-Encoder, der zudem über verschiedene Funktionen zur Videoverarbeitung verfügt [80]. So kann er zum Beispiel Streams aufnehmen und Signale vermischen und bearbeiten [80]. Wie bereits erwähnt wurde, soll die im Rahmen dieser Thesis ermittelte Lösung in der Cloud laufen und somit einen softwarebasierten Encoder nutzen.

6.2. Video-Codecs

Codec steht für ‚coder/decoder' und definiert eine bestimmte Kompressionsmethode beziehungsweise einen bestimmten Kompressionsalgorithmus, mit dem Audio- oder Videodateien komprimiert werden [38]. Das Ziel dabei ist es, Redundanzen in den Daten zu reduzieren [88]. Für das Streaming bedeutet dies, dass die Audio- oder Videosignale in ein bestimmtes Format konvertiert werden, bevor sie über das Netzwerk übertragen werden [38]. Codecs sind nicht mit dem Containerformat zu verwechseln, das lediglich bestimmt, wie die Daten gespeichert werden [39]. Folgende Abbildung zeigt die Funktionsweise eines Codecs:

Abbildung 26: Datenkompression, eigene Darstellung in Anlehnung an [88]

Im ersten Schritt werden die Eingangsdaten mit einem Encoder beziehungsweise Komprimierungsverfahren komprimiert, also in der Größe reduziert [88]. Danach werden die komprimierten Daten über einen bestimmten Kommunikationskanal übertragen und an den Decoder weitergeleitet [88]. Dieser kann schließlich die Originaldaten aus der komprimierten Form wiederherstellen [88].

Generell werden Codecs in verlustbehaftete (*lossy compression*) und verlustfreie (*lossless compression*) Codecs unterschieden [88]. Verlustbehaftete Codecs komprimieren Daten so, dass das Ausgangssignal nach der Kompression nicht mehr 1:1 wiederhergestellt werden kann. Dahingegen können verlustfreie Codecs die Daten so verpacken, dass das Originalsignal danach wiederhergestellt werden kann [32]. Die gewählte Kompressionsmethode hängt unter anderem von der gewünschten Videoqualität und dem verfügbaren Speicherplatz ab [88].

Da verlustfreie Codecs die Daten so komprimieren, dass sie ohne Qualitätsverlust wiederhergestellt werden können, benötigen sie, im Vergleich zu verlustbehafteten Alternativen, meist mehr Speicherplatz. Da Streaming-Anwendungen jedoch auf geringe Latenzen und damit geringe Datenmengen angewiesen sind, kommen verlustfreie Codecs in diesem Bereich nicht zum Einsatz. Daher werden an dieser Stelle lediglich einige verlustbehafteten Codecs vorgestellt, die beim Streaming genutzt werden, darunter die MPEG-Codecs H.264/MPEG-AVC und H.265/MPEG-HEVC sowie die von Google entwickelten Codecs VP8 und VP9.

Einer der meistgenutzten Videocodecs ist der von der ITU-T Video Coding Experts Group und ISO/IEC JTC1 Moving Picture Experts Group (MPEG) entwickelte H.264/MPEG-AVC [89]. Dieser Codec wurde im Jahr 2003 standardisiert und wird von vielen Streaming-Anwendungen wie zum Beispiel Twitch.tv eingesetzt [89]. Er eignet sich demnach gut für das Streaming mit niedrigen Latenzen und wird häufig für HTTP- und WebRTC-basierte Anwendungen, wie HLS, genutzt [90]. Daneben wird er auch für die Videokodierung bei Blu-ray-Disks und Kabelfernsehen eingesetzt [90]. Containerformate, die von H.264 unterstützt werden, sind unter anderem .mp4, .mov, .F4v, .3GP und .ts [90].

Der Nachfolger von H.264/MPEG-AVC ist der im Jahr 2013 standardisierte Codec H.265/MPEG-HEVC [89]. HEVC steht für ‚High-Efficiency Video Coding' und ermöglicht im Gegensatz zu H.264 höhere Kompressionsraten. Dabei kann die Bitrate bei gleicher Qualität um etwa 50 % reduziert werden [89], wofür allerdings auch eine höhere Rechenleistung notwendig ist [89].

VP8 wurde im Jahr 2010 von Google als open-source-Alternative zum H.264-Codec auf den Markt gebracht, um Videostreams zu kodieren, die in WebM-Dateien verpackt werden [91]. Um kompaktere Bitstreams zu erhalten, begann Google Ende 2011 damit, am Nachfolger-Codec VP9 zu arbeiten. Auch dieser ist ein open-source-Codec, der unter anderem von YouTube genutzt wird [89].

6.3. Streaming-Server / Media-Server

Streaming-Server sind dafür verantwortlich, den kodierten Bitstream und entsprechende Metadaten vom Encoder entgegenzunehmen, neu zu verpacken und für Clients bereitzustellen [32]. Sie sind ebenfalls dafür zuständig, die Übertragung zu kontrollieren und die Verfügbarkeit des Streams für den Client sicherzustellen [32]. Dabei können eigene On-Premise-Streaming-Server aufgesetzt und konfiguriert oder sogenannte ‚Live-Streaming-Plattformen' genutzt werden.

Mittlerweile ist eine Vielzahl verschiedener Live-Streaming-Plattformen im Netz vorhanden, die es Nutzern ermöglichen, Live-Videomaterial hochzuladen und in Echtzeit auszustrahlen. Sie werden auch als ‚Video-Hosting-Lösungen' oder ‚Online-Video-Plattformen' bezeichnet [92]. Dazu zählen zum Beispiel die bekannten Plattformen YouTube oder Twitch, aber auch Dacast oder Restream IO.

Online-Video-Plattformen werden nicht nur dazu eingesetzt, Videos zu hosten beziehungsweise Streams zur Verfügung zu stellen. Häufig enthalten sie auch einen Video-Player und eine Möglichkeit, Daten automatisch zu transkodieren und für das adaptive Bitrate-Streaming bereitzustellen [41]. Viele Plattformen bieten außerdem Monetarsierungs- und Werbemöglichkeiten sowie Sicherheitsmechanismen und Möglichkeiten der Zugriffskontrolle [41].

Solche Online-Video-Plattformen verfügen häufig über sogenannte ‚Content-Delivery-Netzwerke', um die Streams auszustrahlen. Ein Content-Delivery-Netzwerk (CDN) ist ein verteiltes Serversystem, das Inhalte an Clients verteilt [93]. Dabei werden Daten von einem Hauptserver auf sogenannte ‚*Surrogate Server*' kopiert, die hierarchisch angeordnet und im Internet verteilt sind [94] (siehe Abbildung 27). Häufig angefragte Inhalte werden von diesen Servern abgefragt und in lokalen Netzwerk-Caches gespeichert. Bei einer Client-Anfrage wird diese an den nächsten Cache weitergeleitet, der für die Bereitstellung des angefragten Inhaltes zuständig ist [94]. Die Idee dahinter ist, dass die Server möglichst nahe am Endnutzer platziert sind, um den Netzwerkverkehr zu reduzieren und möglichst geringe Latenzen zu ermöglichen [94]. Ein solches Netzwerk bietet demnach eine gute Möglichkeit zur Skalierung und zuverlässigen Datenübertragung [93].

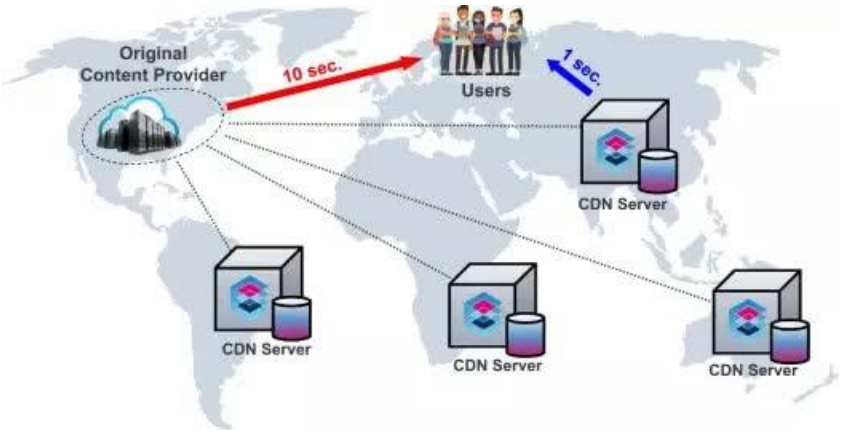

Abbildung 27: Content-Delivery-Netzwerk, Quelle: [93]

Zu den bekannten Content-Delivery-Netzwerken gehören unter anderem *Amazon Web Service* (AWS), *Bunny Stream*, *Cloudflare*, *KeyCDN* und *LimelightNetworks* [93].

Da es im Rahmen dieses Projektes jedoch wichtig ist, unabhängig von Drittanbietern zu bleiben und die volle Kontrolle über den Inhalt und den Zugriff der Inhalte zu haben, werden im Folgenden einige Möglichkeiten betrachtet, um einen eigenen, zentralen Streaming-Server aufzusetzen.

Eine bekannte Möglichkeit ist die open-source-Software *nginx*, bei der es sich um einen Webserver handelt, der unter anderem RTMP- und HLS-Streaming unterstützt [93]. Die Software kann auf einfache Weise auf einem eigenen Server installiert und konfiguriert werden. Eine Konfigurationsdatei ermöglicht das einfache Anpassen der Portnummern sowie verschiedener Server- und Streaming-Optionen. So ist es zum Beispiel möglich, mit dem RTMP-Modul einen eingehenden RTMP-Stream entweder in HLS-Segmente für das HLS-Streaming bereitzustellen oder aber den Stream an eine oder mehrere Streaming-Plattformen wie YouTube oder Twitch.tv zu senden.

Eine weitere Option ist der sogenannte ‚*Red5 Media Server*', bei dem es sich ebenso um eine open-source-Software handelt, um Live-Streaming-Anwendungen zu realisieren [95]. Dieser unterstützt unter anderem das Live-Streaming von Flash, HLS und RTSP [95].

Ant Media ist eine weitere Software-Option, um einen eigenen Streaming-Server zu realisieren. Sie unterstützt unter anderem adaptives Bitrate-Streaming und geringe Latenzzeiten [96]. Zu den unterstützten Streaming-Protokollen gehört zum Beispiel die Protokollsammlung WebRTC, die sich unter anderem zum Streaming von IP-Kamerasignalen, Videokonferenzen, E-Learning-Anwendungen und Videospielen eignet [96]. Im Gegensatz zur bereits genannten Server-Software *nginx* und dem *Red5 Media Server* ist *Ant Media* eine lizenzbasierte Software, die je nach Umfang zu verschiedenen Preisen verfügbar ist [96].

6.4. Video-Player

Zuletzt wird im Rahmen dieses Kapitels zur Marktübersicht ein Blick auf einige bekannte Video-Player geworfen, die genutzt werden können, um einen Stream wiederzugeben.

Ein bekannter Vertreter unter den Streaming-Video-Playern ist der *Adobe Flash Player*, der das Abspielen diverser Flash-Inhalte, wie zum Beispiel das Abspielen von RTMP-Streams, ermöglicht [97]. Seit Januar 2021 sind Flash-Inhalte jedoch nicht mehr im *Adobe Flash Player* abspielbar. Der Video-Player gilt mittlerweile als veraltet. Die meisten Browser unterstützen diesen nicht mehr und er wurde von diversen Alternativen abgelöst [97].

Ein weiterer, weit verbreiteter Videoplayer ist der *VLC Media Player* [98]. Bei diesem handelt es sich um einen kostenlosen, open-source-Videoplayer, der fast alle Codecs abspielen kann und auf allen Plattformen läuft [98]. Es ist sogar möglich, RTMP-Streams mit dem *VLC Media Player* wiederzugeben. Allerdings können Videos nur lokal in diesem Videoplayer abgespielt werden. Um Videos im Browser ansehen zu können, ist ein zusätzliches Plugin notwendig [99].

Der Standard-Videoplayer, der von fast allen Geräten und Browsern unterstützt wird, ist der HTML5-Videoplayer [41]. Dabei können Medieninhalte über die Tags <video> und <audio> direkt in eine Webseite eingebettet werden, ohne dass ein zusätzliches Plugin installiert werden muss [100]. Dieser Videoplayer wird von allen neuen Browsern unterstützt, um Streaming-Inhalte auf Webseiten zur Verfügung zu stellen [100]. Mittlerweile gibt es verschiedene open-source- beziehungsweise kostenpflichtige HTML5-Videoplayer, die verschiedene, zusätzliche Funktionen bieten [100]. Ein bekannter HTML5-Videoplayer ist der *Flowplayer*, der zwar kostenpflichtig ist, dafür aber diverse Analyse- und Monetarisierungsmöglichkeiten bietet. Eine Alternative ist der *JWPlayer*, der für Werbeanzeigen optimiert ist und zudem verschiedene Analyse-Werkzeuge bereitstellt. Verkauft wird dieser als weltweit schnellster HTML5-Videoplayer [101]. Eine kostenfreie Alternative ist der Javascript-basierte *video.js*-Videoplayer, der individualisiert angepasst werden kann. Die genannten Videoplayer unterstützen alle die Wiedergabe von HLS- und MPEG-DASH-Streams [100]. Neben diesen Videoplayern gibt es abhängig von den Anforderungen weitere Möglichkeiten, wie zum Beispiel *hls.js* oder *dash.js* [100].

Wie bereits erwähnt wurde, gibt es außerdem verschiedene Online-Streaming-Plattformen, die neben ihrer Funktion als Streaming-Server teilweise auch eigene Videoplayer mit sich bringen, die in eine Webseite eingebettet werden können, um einen Stream dort zur Verfügung zu stellen. Ein Beispiel ist *StreamingVideoProvider*, eine Online-Streaming-Plattform, die unter anderem einen Videoplayer bietet, der in die eigene Webseite eingebettet werden kann [102]. Dazu liefert die Plattform den Code, der einfach in den Webseiten-Code integriert werden kann. Außerdem kann der Player individuell an das eigene Unternehmen angepasst werden, indem zum Beispiel das eigene Logo eingefügt werden kann [102].

7. Lösungsansätze

Nachdem im letzten Kapitel eine Übersicht verschiedener Streaming-Komponenten präsentiert wurde, geht es in diesem Kapitel um die Vorstellung verschiedener, während der Recherche ermittelter Lösungsansätze für das Projekt. Dabei werden für jeden Ansatz die notwendigen Komponenten und deren Funktionsweise erläutert. Im Anschluss werden verschiedene Vergleichskriterien dargelegt und die Lösungsansätze basierend darauf verglichen, um im letzten Schritt den für das Projekt bestmöglichen Ansatz auszuwählen.

7.1. Ansatz Nr. 1: Integration von NoVNC in *CONNECTED*

Da jeder Rechner auf aktiven Tunnelbohrmaschinen bereits über einen VNC-Server verfügt (siehe 3.1.), ist ein naheliegender Ansatz, einen VNC-Client, der sich mit dem Server auf der TBM verbinden kann, direkt in *CONNECTED* einzubetten, um den Bildschirminhalt zu integrieren. Allerdings handelt es sich bei den meisten VNC-Clients um eigenständige Applikationen, die heruntergeladen werden müssen, um eine Verbindung mit einem VNC-Server aufzubauen und den Desktopinhalt des Servers zu erhalten. Daher wurde nach einer Möglichkeit gesucht, den übertragenen Bildschirminhalt als Stream einzubinden. Schließlich wurde eine Implementierung eines VNC-Clients gefunden, die es ermöglicht, den Client direkt im Browser aufzurufen: *NoVNC* [103].

NoVNC ist ein Remote-Desktop-Client, der auf JavaScript und HTML5 basiert und es ermöglicht, über sogenannte ‚WebSockets' mit einem entfernten Rechner zu kommunizieren. Dabei kann direkt über einen Browser eine Verbindung zu einem entfernten Rechner aufgebaut werden [104]. Der Unterschied zu anderen VNC-Clients besteht darin, dass NoVNC webbasiert ist und sich daher auf simple Weise in andere Applikationen, wie zum Beispiel *CONNECTED*, integrieren lässt [104].

Dieser erste Lösungsansatz schlägt vor, die HTML-Seite von NoVNC direkt in einen HTML-Block in *CONNECTED* einzubetten und damit den VNC-Stream des entsprechenden Servers bereitzustellen. Folgende Abbildung zeigt die Architektur dieses Lösungsansatzes:

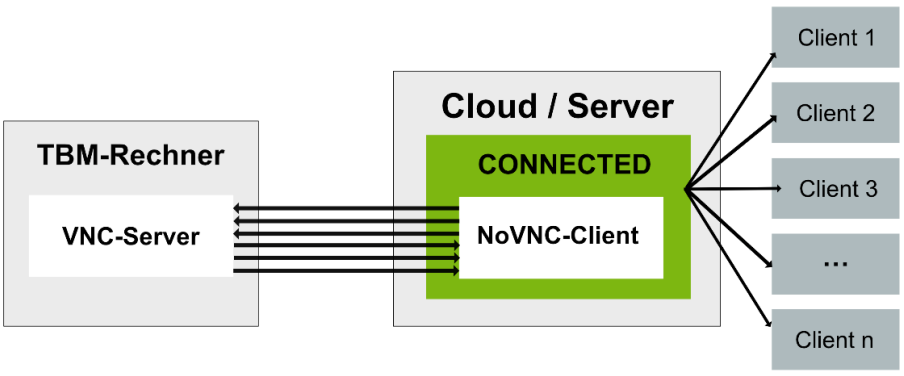

Abbildung 28: Architektur Lösungsansatz Nr. 1, eigene Darstellung

Die Idee ist, dass eine Verbindung des NoVNC-Clients mit dem VNC-Server auf der Tunnelbohrmaschine aufgebaut und der Bildschirm übertragen wird, sobald ein Client *CONNECTED* aufruft. Um zu verhindern, dass der Client den Bildschirm steuern kann, was aus Sicherheitsgründen vermieden werden muss, kann der NoVNC-Client so konfiguriert werden, dass der Inhalt lediglich dargestellt wird, die Eingaben der Clients aber nicht an den VNC-Server zur Steuerung zurückgeschickt werden.

Der Vorteil dieser Lösung ist das einfache Einbetten des NoVNC-Clients in die Webapplikation. Allerdings ist für diese Lösung viel Bandbreite notwendig, da jeder neue Client eine eigene Verbindung zum VNC-Server benötigt und der Datenverkehr im Tunnel dadurch groß wird. Außerdem entsteht ein großes Sicherheitsrisiko, da die Clients eine direkte Verbindung mit dem TBM-Rechner aufbauen.

7.2. Ansatz Nr. 2: VNC-Streaming mit vnc2flv und ffmpeg

Wie zuvor beschrieben wurde (siehe 7.1.), wäre ein optimaler Lösungsansatz die Nutzung der bereits installierten VNC-Server auf den Tunnelbohrmaschinen, um zu verhindern, zusätzliche Software auf den Rechnern installieren zu müssen. Daher wurde ein weiterer Lösungsansatz ermittelt, der die VNC-Technologie nutzt, um den Desktopinhalt der TBM-Rechner in *CONNECTED* als Stream bereitzustellen.

Um das beschriebene Problem des letzten Ansatzes zu verhindern, dass alle Clients eine direkte Verbindung zu den VNC-Servern aufbauen, wurde nach einer Lösung gesucht, die es ermöglicht, den Bildschirminhalt lediglich einmal vom TBM-Rechner in die Cloud zu transportieren und von dort aus an die Clients zu verteilen. Dabei wurde die Software *vnc2flv* gefunden [105], die es ermöglicht, VNC-Sessions aufzunehmen und als Flash-Datei zu speichern. Diese Software wurde während des Testens adaptiert und mit *ffmpeg* kombiniert, um eine Streaming-Lösung zu erhalten. Folgende Abbildung zeigt die Architektur dieses Ansatzes:

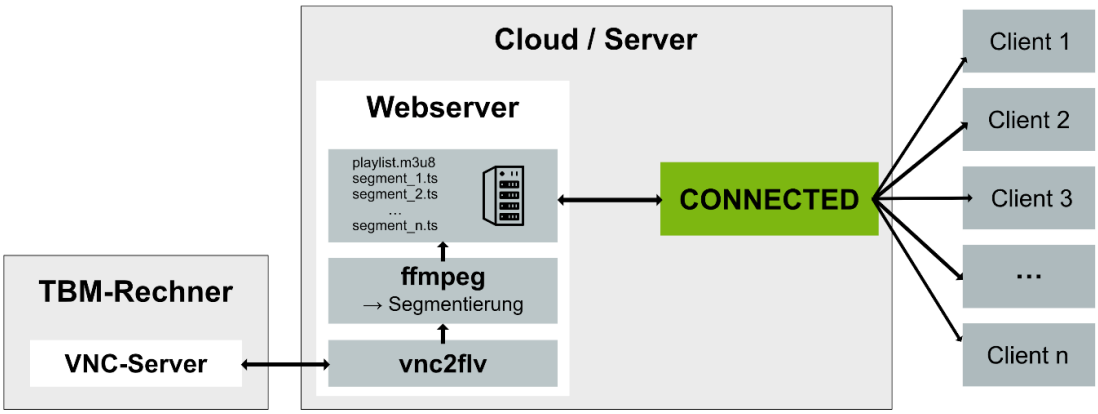

Abbildung 29: Architektur Lösungsansatz Nr. 2, eigene Darstellung

Die Software *vnc2flv* basiert auf Python und ermöglicht es, VNC-Sessions aufzunehmen und diese in Flash-Dateien (.flv) zu speichern [105]. Die Idee bei diesem Ansatz besteht darin, den Stream nicht in einer Datei zu speichern, sondern kontinuierlich an *ffmpeg* weiterzuleiten und per HLS auf *CONNECTED* zur Verfügung zu stellen. Dazu soll *ffmpeg* den Stream von *vnc2flv* als Eingangssignal entgegennehmen, in Segmente aufteilen und auf einem Webserver in Form von HLS-Chunks bereitstellen. Clients können dementsprechend die Segmente herunterladen und wiedergeben (siehe 5.4.3).

7.3. Ansatz Nr. 3: Integration einer Bildschirmaufnahme

Ein weiterer Lösungsansatz ist es, den Bildschirminhalt des TBM-Rechners mittels einer Software als Stream aufzunehmen und diesen an einen Medien-Server zu schicken, der sich auf dem Webserver in der Cloud befindet. Der Medien-Server ist im nächsten Schritt dafür verantwortlich, den Stream an die Clients zu verteilen. Folgende Abbildung zeigt die notwendigen Komponenten für diesen Ansatz:

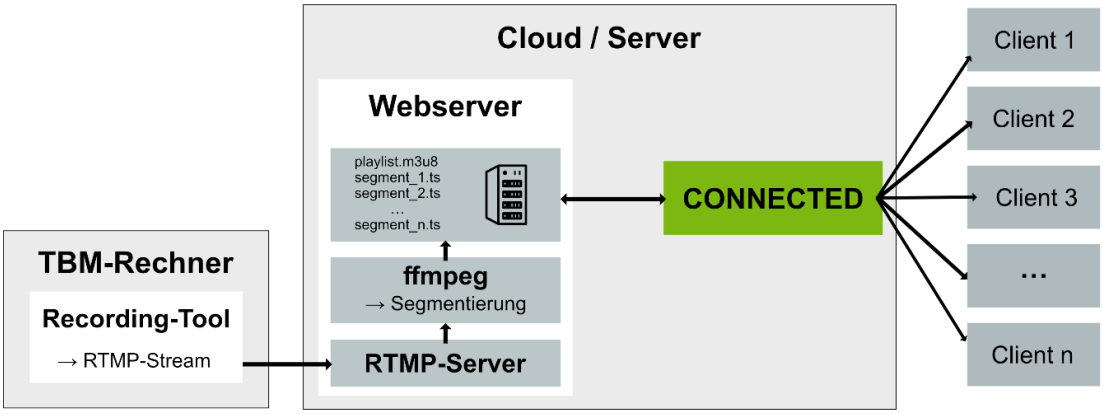

Abbildung 30: Architektur Lösungsansatz Nr. 3, eigene Darstellung

Für die Bildschirmaufnahme wird im ersten Schritt eine Recording-Software benötigt, die eine Aufnahme des TBM-Bildschirms macht. Diese Software muss in der Lage sein, die Aufnahme als Stream an einen Streaming-Server zu übertragen, damit der Server den Stream für die abrufenden Clients bereitstellen kann.

Die Idee ist, dass auf dem TMB-Rechner ein Skript läuft, das den Bildschirm aufnimmt und als RTMP-Stream in die Cloud schickt. Dort soll der RTMP-Stream in Segmente zerlegt werden, um per HLS an die Clients verteilt zu werden. Dazu muss sich ein RTMP-Server in der Cloud befinden, der den Stream vom TBM-Rechner entgegennimmt. In der Cloud muss ein Skript laufen, das den RTMP-Stream in abrufbare Segmente zerlegt, die von Clients heruntergeladen und wiedergegeben werden können.

Das Problem bei dieser Architektur ist, dass die TBM-Rechner in der Lage sein müssen, RTMP-Streams aus dem internen Netzwerk ins Internet zu schicken, damit der RTMP-Server in der Cloud diesen weiterverarbeiten kann. Da TBM-Rechner aus Sicherheitsgründen keine Inhalte ins Internet schicken

können, musste die Architektur von Abbildung 30 angepasst werden, um diesen Lösungsansatz zu realisieren. Folgende Abbildung zeigt die angepasste Architektur.

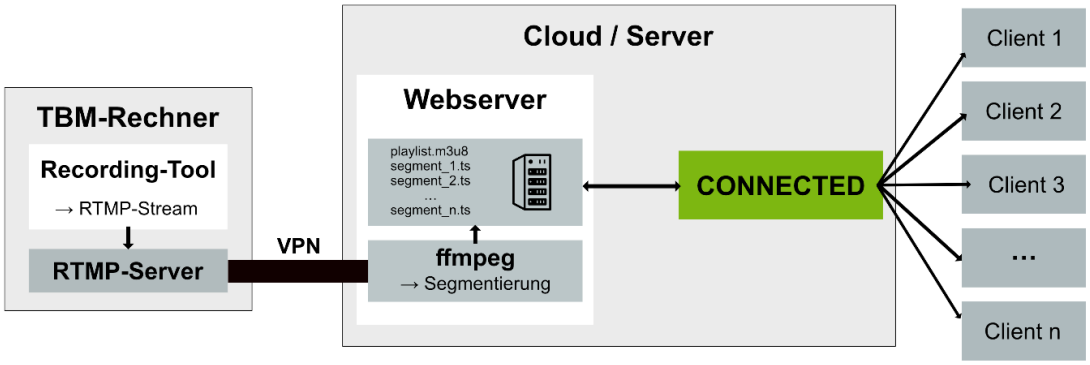

Abbildung 31: Angepasste Architektur Lösungsansatz Nr. 3, eigene Darstellung

Um zu vermeiden, dass die TBM-Rechner den RTMP-Stream ins Internet schicken müssen, wurde die Architektur angepasst, sodass die Verbindung zwischen dem Webserver in der Cloud und dem TBM-Rechner per VPN (virtuelles privates Netzwerk) erfolgt. Dabei wird ein RTMP-Server auf dem TBM-Rechner installiert, der in einer virtuellen Linux-Umgebung läuft. Dadurch kann das Skript beziehungsweise die Aufnahmesoftware den RTMP-Stream an den RTMP-Server auf demselben Rechner schicken. In der Cloud läuft weiterhin ein Webserver, allerdings ohne RTMP-Server. Auf diesem läuft ein weiteres Skript, das den RTMP-Stream per VPN-Verbindung vom TBM-Rechner abholt und in HLS-Segmente zerlegt. Diese können dann von Clients abgerufen werden.

7.4. Ansatz Nr. 4: Peer-to-Peer-Streaming mit WebRTC

Letztlich wurde ein Lösungsansatz ermittelt, der auf der Protokollsammlung *WebRTC* basiert und auf der Peer-to-Peer-Kommunikation aufbaut. Die Idee dabei ist es, die vorhandenen VNC-Server auf den Tunnelbohrmaschinen beizubehalten und auf dem Webserver in der Cloud jeweils ein VNC-Client zu installieren, um den Bildschirminhalt von der TBM auf den Server zu übertragen. Im nächsten Schritt soll der VNC-Client, zum Beispiel NoVNC, in eine Seite integriert werden, die im Browser aufgerufen werden kann. Im Anschluss kann WebRTC eingesetzt werden, um die Browser-Browser-Kommunikation zwischen zwei Peers zu ermöglichen, wobei es sich einmal um den Browser mit dem VNC-Client und einmal um *CONNECTED* handelt. Folgende Abbildung zeigt die Architektur dieses Lösungsansatzes:

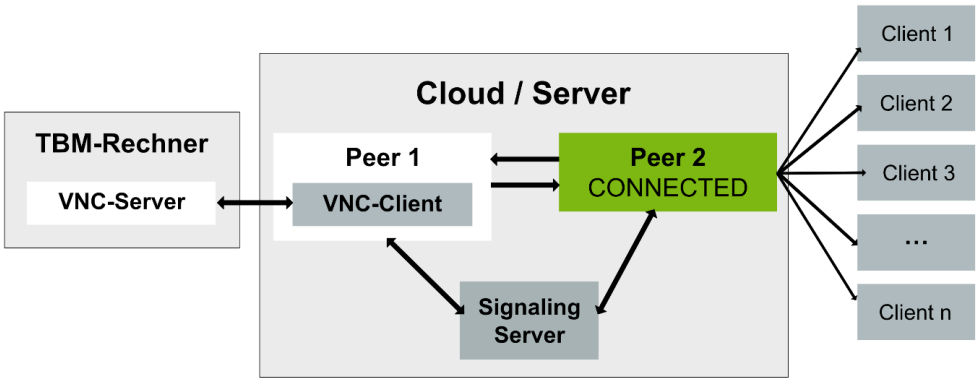

Abbildung 32: Architektur Lösungsansatz Nr. 4, eigene Darstellung

Neben dem VNC-Client, der in der Cloud installiert sein muss, wird ein Signaling-Server benötigt, um die Verbindung zwischen Peer 1 und Peer 2 aufzubauen. Signaling wird benötigt, um drei verschiedene Informationen auszutauschen. Einerseits können Kontrollnachrichten ausgetauscht werden, um die Kommunikation zu initialisieren oder zu beenden beziehungsweise Fehler zu melden [76]. Darüber hinaus wird der Signaling-Server benötigt, um Netzwerkinformationen zu erhalten und die verfügbaren Codecs und Auflösungen zu ermitteln [76]. Sobald der Informationsaustausch mit dem Signaling-Server abgeschlossen ist, können Peers direkt miteinander kommunizieren.

Wenn sich Peers in verschiedenen IP-Netzwerken befinden, wird zusätzlich ein sogenannter ‚STUN-Server' (Session Traversal Utilities for NAT) oder ‚TURN-Server' (Traversal Using Relays around NAT) benötigt, um die Kommunikation zwischen Browsern zu ermöglichen [106]. Dies ist notwendig, wenn sich Peers hinter NAT-Gateways befinden. Da sich die Peers bei diesem Lösungsansatz im selben Netz auf dem Server befinden, kann darauf verzichtet werden.

Ein großer Vorteil dieses Ansatzes ist die Tatsache, dass kein traditioneller Streaming-Server benötigt wird, um Daten auszutauschen. Hinzu kommt, dass WebRTC aufgrund der Peer-to-Peer-Verbindungen geringe Latenzen bietet. Die Inhalte können direkt zwischen Peers ausgetauscht werden und müssen nicht von einem Streaming-Server zwischengespeichert werden, weshalb der Bildschirminhalt der TBM-Rechner nahezu in Echtzeit als Stream in *CONNECTED* zur Verfügung gestellt werden könnte. Der Unterschied zum ersten Lösungsansatz besteht außerdem darin, dass nur eine VNC-Verbindung zwischen Peer 1 und dem VNC-Server auf der TBM aufgebaut werden muss und keine direkte Verbindung zwischen Clients und den TBM-Rechnern notwendig ist.

Ein Nachteil des Protokolls ist jedoch, dass die Kommunikation unter Browsern nicht immer den Sicherheitsstandards eines Unternehmens entspricht [106]. Die Signalisierungsverarbeitung ist nicht standardisiert, sodass jede WebRTC-Anwendung verschiedene Sicherheitsprotokolle einsetzen kann [106]. Beim Verbindungsaufbau zwischen Browsern mit WebRTC muss außerdem beachtet werden, dass die Datenschutzstandards nicht immer gegeben sind [106].

7.5. Vergleich der Lösungsansätze

Nachdem die vier beschriebenen Lösungsansätze ermittelt wurden, werden die Ansätze anhand verschiedener Kriterien und Anforderungen verglichen, um schließlich die bestmögliche Streaming-Lösung für das Projekt zu wählen. Die gewählten Vergleichskriterien wurden in Anlehnung an die Projektanforderungen bestimmt, die am Anfang dieser Thesis definiert wurden (siehe 3.). Sie werden zudem mit einer Skala von einem bis drei Sternen (*, **, ***) gewichtet, um die Priorisierung der Kriterien hervorzuheben (siehe Tabelle 3).

Das erste Kriterium für den Vergleich ist, dass die gewählte Streaming-Lösung eine geringe Latenz von unter 20 Sekunden bieten sollte, um möglichst nahe an das Echtzeit-Streaming zu kommen. Hinzu kommt, dass der übertragene Stream eine gute Qualität bieten soll. Damit ist eine Auflösung von mindestens Full HD gemeint, also 1920 x 1080 Pixeln, und eine Framerate von 20 bis 30 Bildern pro Sekunde. Weiterhin soll eine direkte Verbindung zwischen den Clients und den TBM-Rechnern verhindert werden, um zu vermeiden, dass Clients Zugriff darauf erhalten, was ein Risikopotenzial bergen würde. Zudem soll die Lösung möglichst sicher sein, um zu verhindern, dass unberechtigte Nutzer Zugriff auf die übertragenen Daten erhalten. Ein weiterer Vorteil wäre es, wenn keine weitere Software auf den TBM-Rechnern installiert werden müsste, sodass die bereits vorhandenen VNC-Server eingesetzt werden können. Letztlich soll die Lösung mit allen TBM-Rechnern kompatibel sein, um den Lösungsansatz in der Zukunft für alle aktiven Maschinen anbieten zu können.

Die folgende Tabelle gibt eine Übersicht über die Lösungsansätze und darüber, welche der genannten Kriterien mit dem Ansatz erfüllt werden und welche nicht. Dabei bedeutet das Symbol ✓, dass die Lösung das entsprechende Kriterium erfüllt, und das Symbol ✗, dass der Ansatz das Kriterium nicht erfüllt.

	NoVNC (1)	vnc2flv / ffmpeg (2)	Bildschirmaufnahme TBM-Rechner (3)	WebRTC (4)
Geringe Latenz **	✓	✓	✓	✓
Auflösung (mind. Full HD) **	✓	✓	✓	✓
Framerate von >= 20 fps **	✗	✗	✓	✗
Keine direkte Verbindung zw. Client-TBM-Rechner ***	✗	✓	✓	✓
Sicherheit ***	✗	✓	✓	✗
Keine zusätzliche Software auf TBM-Rechnern *	✓	✓	✗	✓
Kompatibilität mit allen TBM-Rechnern ***	✓	✗	✓	✓

Tabelle 3: Vergleich der Lösungsansätze anhand Kriterien, eigene Darstellung

Wie die Tabelle zeigt, konnte kein Lösungsansatz gefunden werden, der alle Kriterien zu 100 % erfüllt. Daher wurden die Nachteile aller Ansätze abgewogen, um die bestmögliche Lösung zu finden.

Der erste Ansatz (1), der auf der NoVNC-Software basiert, konnte sofort ausgeschlossen werden. Grund dafür ist das hohe Sicherheitsrisiko, das besteht, wenn Clients eine direkte Verbindung zu den TBM-Rechnern aufbauen. Außerdem würde dieser Ansatz zu viel Bandbreite benötigen, da der Stream bei jedem neuen Client erneut vom TBM-Rechner in die Cloud transportiert werden müsste. Da ein hoher Datenverkehr auf den Leitungen in den Tunnel-Netzwerken vermieden werden soll, war dieser Ansatz nicht geeignet. Ein weiterer Nachteil ist, dass die VNC-Technologie auf dem *Remote-Framebuffer*-Protokoll basiert und die Framerate beziehungsweise die Update-Rate von der Netzwerkgeschwindigkeit abhängig und demnach nicht konstant ist.

Die Software *vnc2flv*, die für den zweiten Lösungsansatz (2) zum Einsatz kommt, war zunächst vielversprechend. Deren Vorteile bestehen darin, dass sie die bereits installierten VNC-Server der TBM-Rechner nutzt und es ermöglicht, VNC-Sessions direkt aufzunehmen und zu streamen. Da die VNC-Technologie, die nicht nur für diesen, sondern auch für Lösungsansatz (1) und (4) zum Einsatz kommt, den Bildschirm in Form eines Arrays von Pixelwerten überträgt, entspricht die Auflösung der Aufnahme der ursprünglichen Bildschirmauflösung des jeweiligen entfernten Rechners. Da die TBM-Rechner meist eine Auflösung von 1920 x 1080 (Full HD) oder 3840 x 2160 Pixel haben (siehe 3.1.), ist die Anforderung an eine gute Auflösung des Videostreams erfüllt. Dahingegen kann, wie bereits zuvor genannt wurde, aufgrund der VNC-Technologie keine konstante Framerate sichergestellt werden.

Ein schwerwiegendes Problem bei diesem Ansatz ist außerdem die fehlende Kompatibilität zu modernen VNC-Servern. Da die Software sowie die VNC-Technologie alt sind, konnte dieser

Lösungsansatz lediglich in einer Testumgebung für einen VNC-Server erfolgreich zum Laufen gebracht werden. Bei der Verbindung der Software zu neueren VNC-Servern lieferte *vnc2flv* keinen korrekten Stream. Aus den genannten Gründen musste auch dieser Ansatz, trotz der gegebenen Vorteile, ausgeschlossen werden.

Ein großer Vorteil des vierten Lösungsansatzes (4) mit WebRTC sind die geringen Latenzzeiten, die ermöglicht werden. Außerdem baut der Ansatz auf den vorhandenen VNC-Servern auf und vermeidet durch die Peer-to-Peer-Architektur den direkten Zugriff von Clients auf die TBM-Rechner. Insbesondere das Sicherheitsrisiko aufgrund der genannten Peer-to-Peer-Verbindungen spricht jedoch gegen diesen Lösungsansatz. Da es sich bei den übertragenen Bildschirminhalten der TBM-Rechner um sensible Daten handelt, die über direkte Verbindungen zwischen Browsern ausgetauscht werden, können die Datenschutzrichtlinien nicht erfüllt werden. Aus diesen Gründen wurde dieser Ansatz ausgeschlossen.

Letztlich blieb Lösungsansatz (3), der darauf basiert, den Bildschirm der TBM-Rechner aufzunehmen und als Stream an einen RTMP-Server zu senden. Er kombiniert das Streaming mit RTMP und HLS, um abrufbare Streaming-Segmente für Clients bereitzustellen. Der Vorteil dieses Ansatzes ist vor allem die hohe Sicherheit, da weder VNC-Verbindungen zwischen Clients und TBM-Rechnern noch Peer-to-Peer-Verbindungen aufgebaut werden müssen, um das Streaming zu ermöglichen. Weiterhin können mit diesem Ansatz die Videoauflösung und eine Framerate von 20 bis 30 fps bei der Aufnahme sichergestellt werden. Auch eine niedrige Latenz (von unter 20 Sekunden) kann realisiert werden. Allerdings können die bereits vorhandenen VNC-Server auf den Tunnelbohrmaschinen nicht verwendet werden, sodass zusätzliche Software auf den TBM-Rechnern installiert werden muss. Dennoch ist der Installationsaufwand der zusätzlichen Software minimal (siehe 8.) und die Vorteile des Ansatzes überwiegen diesen Nachteil. Daher wurde dieser Ansatz für den Prototyp gewählt. Die Implementierung wird in den nächsten Kapiteln näher erläutert.

8. Implementierung Prototyp

In diesem Kapitel geht es um die Implementierung eines Prototyps, um den gewählten Lösungsansatz für das Streaming in *CONNECTED* zu realisieren. Dazu wird zunächst die verwendete Software näher vorgestellt und die notwendigen Installationsschritte werden beschrieben. Weiter wird erläutert, welche Schritte vorgenommen wurden, um die Kommunikation der Software untereinander zu lösen und schließlich einen funktionsfähigen Prototyp zu erhalten, der den Bildschirminhalt von einem Testrechner quasi in Echtzeit in einem HTML-Videoplayer darstellt, der in der Zukunft in *CONNECTED* integriert werden kann.

Folgende Abbildung zeigt die Systemarchitektur, die in den folgenden Abschnitten umgesetzt wird:

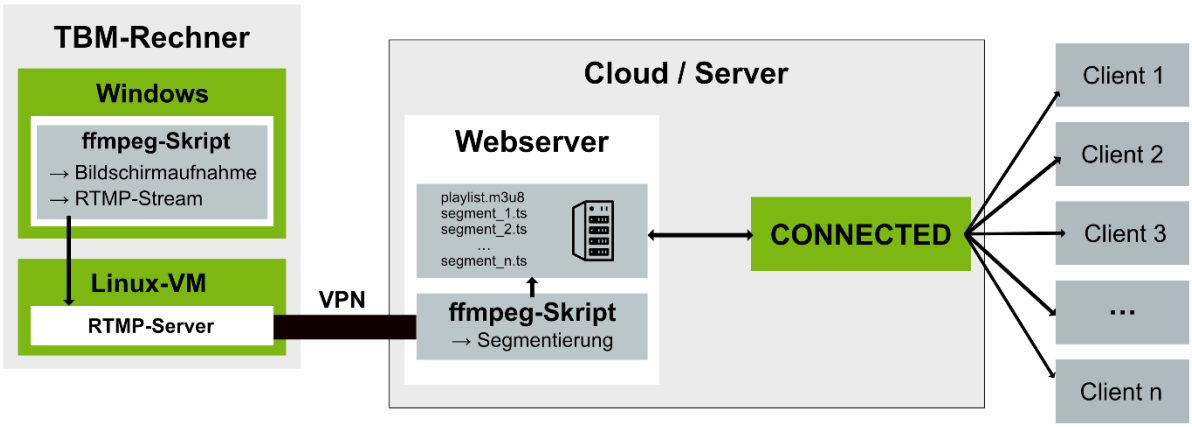

Abbildung 33: Systemarchitektur-Prototyp, eigene Darstellung

Das Ziel ist es, auf den TBM-Rechnern ein Skript laufen zu lassen, das jeweils den Bildschirminhalt mit dem Recorder-Tool *UScreenCapture* beziehungsweise *gdigrab* aufnimmt und mittels *ffmpeg* als RTMP-Stream an einen RTMP-Server sendet. Dieser soll sich auf demselben Rechner innerhalb einer virtuellen Linux-Umgebung befinden.

Um den Stream in die Cloud zu transportieren, muss im ersten Schritt eine verschlüsselte VPN-Verbindung zwischen Webserver und dem TBM-Rechner aufgebaut werden. Schließlich kann ein zweites Skript auf dem Webserver den RTMP-Stream mithilfe von *ffmpeg* in HLS-Videosegmente (Chunks) zerlegen und auf dem Server ablegen. Die aufrufenden Clients können dann die dabei generierte HLS-Playlist vom Webserver anfordern und die Segmente herunterladen und abspielen. Der Prototyp basiert auf den Streaming-Protokollen RTMP und HLS (siehe 5.4.2. & 5.4.3.).

Für Testzwecke werden alle Komponenten auf einem lokalen Windows-Rechner installiert. Hinzu kommt, dass lediglich ein Webserver mit RTMP-Funktionalität installiert wird, der die Funktion des RTMP-Servers auf dem TBM-Rechner und die Funktion des Webservers in der Cloud übernimmt.

8.1. Installation der Software

Im ersten Schritt wird auf die verschiedenen Softwarekomponenten und deren Installationsschritte eingegangen, die für die Streaming-Lösung notwendig sind. Dazu zählen das Aufnahmetool *UScreenCapture*, das Kommandozeilentool *ffmpeg* zur Kompression der Aufnahme und der RTMP- beziehungsweise HTTP-Webserver *nginx*.

8.1.1. ffmpeg

Um das Skript auf den TBM-Rechnern laufen zu lassen, wird *ffmpeg* benötigt (siehe 6.1.2.). Da es sich bei den Rechnern auf den Tunnelbohrmaschinen hauptsächlich um Windows-Rechner handelt, wird an dieser Stelle erläutert, wie *ffmpeg* auf dem Windows-Betriebssystem installiert wird.

Im Prinzip reicht es aus, die *ffmpeg*-Dateien herunterzuladen und im entsprechenden Ordner abzulegen. Allerdings müssen die *ffmpeg*-Kommandos in diesem Ordner abgesetzt werden, um Zugriff darauf zu haben. Daher wird zusätzlich die Umgebungsvariable „Path“ auf dem Windows-Rechner um *ffmpeg* erweitert, sodass *ffmpeg*-Kommandos von jedem beliebigen Speicherort auf dem Rechner aufgerufen werden können.

Im ersten Schritt wird die gewünschte Software-Version von der *ffmpeg*-Seite [107] heruntergeladen. Dabei sollte darauf geachtet werden, dass es sich um eine stabile Version und das korrekte Betriebssystem handelt. Nachdem der Zip-Ordner heruntergeladen wurde, wird er entpackt und geöffnet. Im Anschluss kann ein neuer Ordner (*ffmpeg*) auf dem C-Laufwerk des Windows-Rechners angelegt werden und die Dateien *ffmpeg.exe*, *ffplay.exe* und *ffprobe.exe* können in den neu angelegten Ordner kopiert werden.

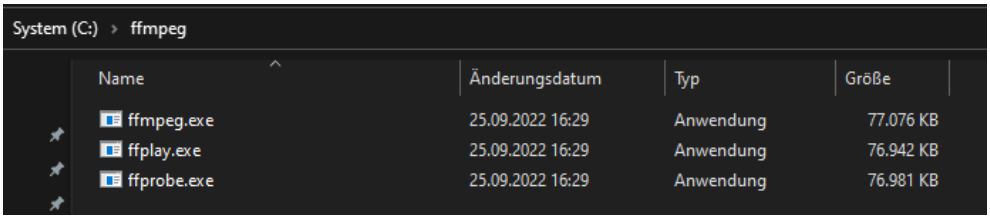

Abbildung 34: ffmpeg-Dateien, eigener Screenshot

Im nächsten Schritt wird die Umgebungsvariable „Path“ um den Speicherort der *ffmpeg*-Dateien erweitert. Dazu wird zunächst die Systemsteuerung mit „Systemumgebungsvariablen bearbeiten“ geöffnet.

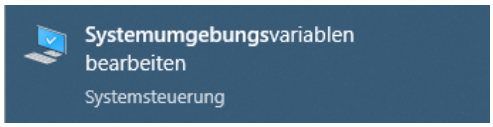

Abbildung 35: Systemumgebungsvariablen bearbeiten, eigener Screenshot

Es öffnet sich folgendes Fenster:

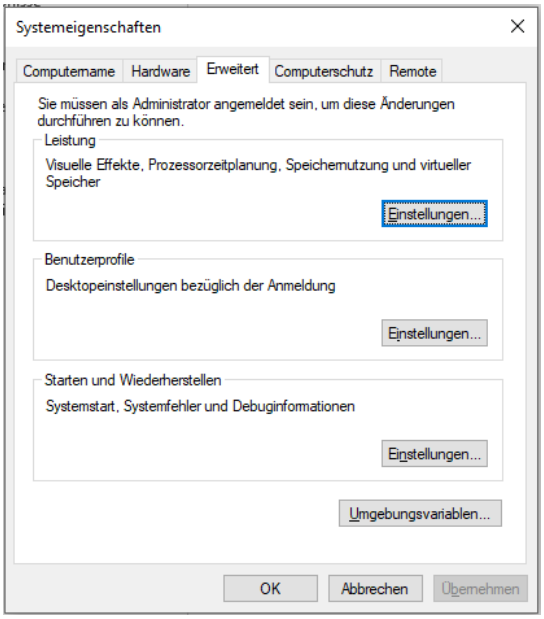

Abbildung 36: Systemeigenschaften, eigener Screenshot

Nun können die definierten Umgebungsvariablen eingesehen und bearbeitet werden, indem auf „Umgebungsvariablen…“ geklickt wird. Dabei öffnet sich das folgende Fenster mit allen definierten Variablen.

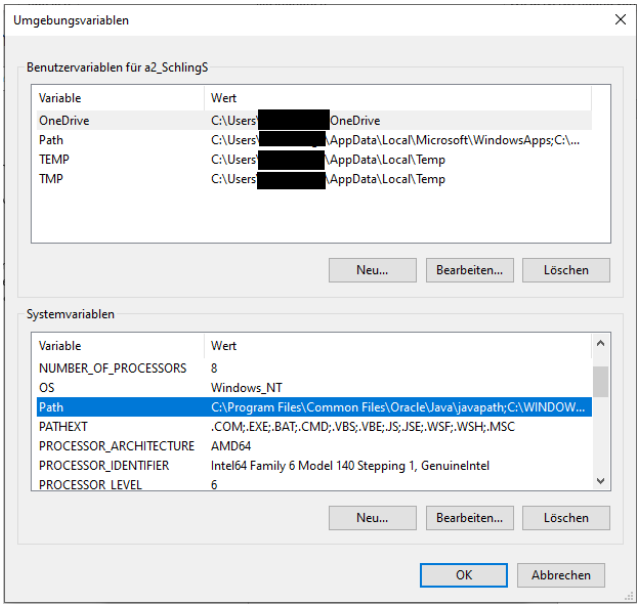

Abbildung 37: Umgebungsvariablen anpassen, eigener Screenshot

Nun kann unter den Systemvariablen die Variable „Path“ (siehe Abbildung 37) ausgewählt und auf „Bearbeiten…“ geklickt werden, um sie zu erweitern. Es öffnet sich das folgende Fenster:

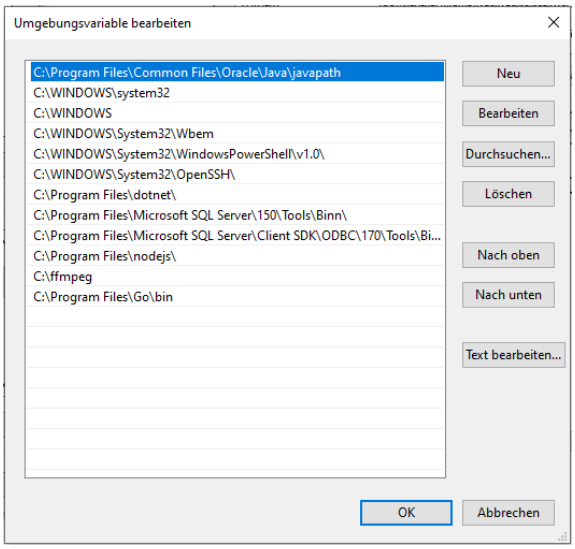

Abbildung 38: ffmpeg-Speicherort ergänzen, eigener Screenshot

Mit Klick auf „Neu“ wird ein neuer Eintrag mit dem Pfad zu den heruntergeladenen *ffmpeg*-Dateien auf dem C-Laufwerk hinzugefügt. Dazu wird lediglich der entsprechende Pfad, in diesem Fall *C:\ffmpeg*, eingetragen. Dies bewirkt, dass *ffmpeg* von überall aus auf dem Rechner aufgerufen werden kann.

Um zu prüfen, ob die Installation erfolgreich war, kann ein neues Kommandozeilenfenster geöffnet und das Kommando *ffmpeg -version* abgesetzt werden. Wird die entsprechende *ffmpeg*-Version ausgegeben, war die Installation erfolgreich. Ansonsten muss der Rechner eventuell neu gestartet werden, um die Änderungen zu übernehmen.

8.1.2. UScreenCapture

UScreenCapture ist eine Software, die es ermöglicht, den Bildschirminhalt eines Rechners aufzunehmen. Um diese für das genannte Skript einzusetzen, muss die Software [108] heruntergeladen und entpackt werden. Durch Klick auf die enthaltene *.msi*-Datei und das Befolgen der Installationsschritte wird sie auf dem Rechner installiert.

Um zu prüfen, ob die Software korrekt installiert und einsatzbereit ist, kann mittels *ffmpeg* eine Liste aller verfügbaren Eingabegeräte ausgegeben werden. Dazu wird folgendes Kommando abgesetzt:

```
ffmpeg -list_devices true -f dshow -i dummy
```

Abbildung 39: ffmpeg-Eingabegeräte auflisten, eigener Screenshot

Bei den Eingabegeräten kann es sich um diverse Geräte zur Aufnahme von Audio- oder Videoinhalten handeln. Darunter zählen zum Beispiel auch in den Rechner eingebaute Kameras und Mikrofone. War die Installation von *UScreenCapture* erfolgreich, erscheint dies als verfügbares Eingabegerät in der Liste.

Abbildung 40: UScreenCapture-Eingabegerät, eigener Screenshot

Alternativ zu *UScreenCapture* kann auch das Aufnahmetool *gdigrab* eingesetzt werden. Dieses ist jedoch standardmäßig unter Windows verfügbar und muss daher nicht explizit installiert werden.

8.1.3. Nginx mit RTMP-Modul

Für den *nginx*-Webserver ist ein RTMP-Modul vorhanden, das es erlaubt, RTMP- und HLS-Streaming auf dem eigenen Webserver zu nutzen. In diesem Fall wird ein *nginx*-Webserver mit RTMP-Modul auf dem lokalen Windows-Rechner installiert. Wie bereits erwähnt wurde, werden der RTMP-Server und der Webserver für Testzwecke vereint, sodass lediglich ein *nginx*-Webserver mit RTMP-Funktionalität installiert wird, der den RTMP-Stream entgegennimmt und gleichzeitig die HLS-Segmente speichert, die vom zweiten Skript erstellt werden. In der endgültigen Lösung wird der RTMP-Server jedoch auf den TBM-Rechnern und ein HTTP-Webserver in der Cloud installiert. Dies ist notwendig, da es den TBM-Rechnern nicht möglich ist, Daten ins Internet zu senden. Dadurch kann das Skript in der Cloud den RTMP-Stream vom RTMP-Server abholen und in HLS-Chunks zerlegen.

Um den Server lokal zu installieren, wird im ersten Schritt die *nginx*-Software mit dem RTMP-Modul heruntergeladen [109] und ein neuer Ordner *nginx* im C-Laufwerk des lokalen Rechners angelegt. Die heruntergeladene Server-Software kann anschließend in diesen Ordner verschoben werden. Im nächsten Schritt kann die Konfigurationsdatei *nginx.conf* angepasst werden, die unter anderem die Portnummern für den Server enthält. Folgende Abbildung zeigt einen Ausschnitt aus der *nginx.conf*-Datei.

```
rtmp {
    server {
        listen 1935;
        application live {
            live on;
            record off;
        }
    }
}

http {
    include mime.types;
    server {
        listen 8080;
        location / {
            root /nginx/srv/;
            index index.html;
            add_header Cache-Control no-cache; # Disable cache
        }
    }
}
```

Abbildung 41: nginx.conf-Datei, eigener Screenshot

In diesem Beispiel fungiert der Server als RTMP-Server und gleichzeitig als Webserver für das HLS-Streaming, weshalb beide Server in der Datei konfiguriert werden. So können für jeden Server die Portnummer, auf die der Server hört, sowie ein Streaming-Endpunkt beziehungsweise das Root-Verzeichnis angegeben werden.

Der RTMP-Server hört in diesem Fall auf den Port 1935 und der Streaming-Endpunkt, an den der RTMP-Stream gesendet wird, heißt *live*. Mit *live on;* wird definiert, dass es sich um einen Live-Stream handelt, der hier empfangen wird, und *record off;* bestimmt, dass der Stream nicht aufgenommen wird.

Der HTTP-Server hört hingegen auf Port 8080 und das root-Verzeichnis befindet sich im Ordner *srv*. Dieser Ordner kann im Ordner *nginx*, in dem sich die Webserver-Software befindet, angelegt werden, um in einem späteren Schritt zum Beispiel eine HTML-Datei und diverse CSS- und JavaScript-Dateien bereitzustellen. Außerdem kann an dieser Stelle ein Ordner *hls* für die erstellten HLS-Segmente angelegt werden.

Der Server kann durch Doppelklick auf die Datei *nginx.exe* gestartet werden und läuft dann im Hintergrund. Dies kann durch Öffnen des Task-Managers auf dem Rechner geprüft werden. Sobald der Server läuft, kann dieser RTMP-Streams entgegennehmen sowie eine HTML-Seite hosten, die im Ordner *srv* abgelegt wurde. Dazu kann in diesem Fall *http://localhost:8080* im Browser aufgerufen werden. Läuft der Webserver, wird entweder die *nginx*-Startseite oder eine eigene HTML-Seite angezeigt, die zuvor im entsprechenden Ordner abgelegt wurde.

8.2. Implementierung

Nachdem die benötigte Software installiert wurde, können die *ffmpeg*-Skripte geschrieben werden. Dazu wird ein Skript benötigt, das die Bildschirmaufnahme startet und den Stream an den RTMP-Server schickt. Ein weiteres Skript empfängt den RTMP-Stream und zerlegt diesen in einzelne Videosegmente. Dabei handelt es sich um zwei Batch-Skripte, die jeweils die notwendigen Befehle beinhalten und per Doppelklick auf die Datei gestartet werden können.

8.2.1. Skript 1: Aufnahme des Bildschirminhalts

Das erste Skript nutzt *ffmpeg* und *UScreenCapture*, um den Bildschirminhalt des Rechners aufzunehmen und per RTMP-Stream an den RTMP-Server zu schicken. Dabei handelt es sich um einen sogenannten ‚*RTMP ingest*' (siehe 5.4.2.). Das .bat-Skript enthält folgendes Kommando:

```
ffmpeg -f dshow -i video="UScreenCapture" -preset fast -tune zerolatency
-keyint_min 12 -sc_threshold 0 -r 24 -g 24 -f flv rtmp://localhost/live/stream1
```

Abbildung 42: Skript 1 zur Bildschirmaufnahme, eigner Screenshot

Im ersten Schritt wird das Eingangssignal für *ffmpeg* definiert. Mit *-f dshow* wird bestimmt, dass es sich um ein Input-Signal eines sogenannten ‚directshow'-Gerätes auf dem Windows-Rechner handelt [110]. directshow-Geräte können jegliche Audio- oder Videoeingabegeräte sein, die mit dem Windows-Rechner verbunden sind. Nachdem *UScreenCapture* erfolgreich installiert wurde, wird die Software als virtuelles Eingabegerät erkannt und kann schließlich mit *-i video="UScreenCapture"* als Videoeingangssignal definiert werden.

Nun werden verschiedene Einstellungen definiert, die bestimmen, wie *ffmpeg* das Eingangssignal verarbeitet und ausgibt. *-preset* bestimmt die Geschwindigkeit, mit der das Signal kodiert wird, im Verhältnis zur Kompressionsrate [111]. Optionen für *-preset* sind zum Beispiel *slow*, *medium*, *fast* und *superfast*. Dabei ist die Qualität im Verhältnis zur Dateigröße besser, je niedriger die gewählte *-preset*-Option ist [111]. Hier wird die Option *fast* gewählt. Daraufhin wird mit *-tune zerolatency* angegeben, dass die Kodierung schnell ablaufen und die Latenz beim Streaming möglichst gering sein soll [111]. *-keyint_min 12* definiert, dass alle zwölf Frames ein Keyframe gesetzt wird. In der Regel arbeitet *ffmpeg* mit einer Szenenerkennung, sodass immer dann ein I-Frame gesetzt wird, sobald *ffmpeg* den Start einer neuen Szene erkennt [112]. Da diese Funktion für die Übertragung des Bildschirminhaltes der TBM-Rechner irrelevant ist, wird dies mit *-sc_threshold 0* ausgeschaltet.

-r gibt die gewünschte Framerate des Outputs an, hier 24 Frames pro Sekunde, und *-g* setzt die sogenannte *Group Picture Size* [112], in diesem Fall ebenso auf 24. Als *Group of Pictures* wird bei der MPEG-Kompression eine Sequenz unterschiedlich komprimierter Bilder bezeichnet. Dabei werden I-Frames, B-Frames und P-Frames eingesetzt, die jeweils mit einer anderen Kompressionsmethode komprimiert und in einer bestimmten Reihenfolge angeordnet werden. Zusammen werden sie als *Group of Pictures* (GOP) bezeichnet [112].

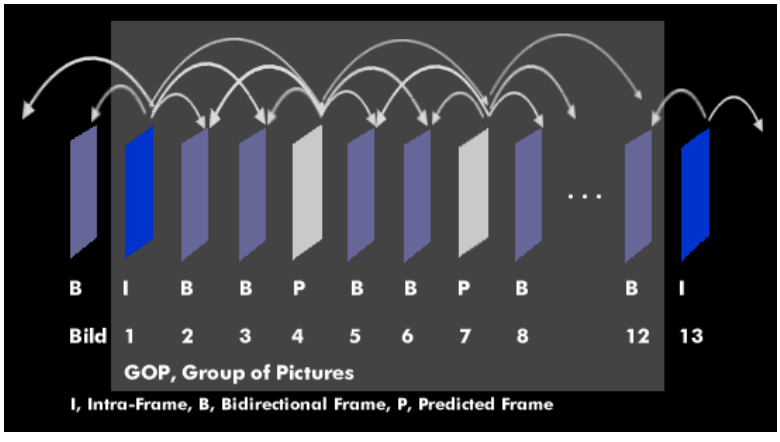

Abbildung 43: Group of Pictures (GOP), Quelle: [113]

I-Frames oder auch Intra-kodierte Bilder sind eigenständige Frames, die auf ähnliche Weise wie ein JPEG-Bild kodiert werden. Sie werden auch als ‚Referenzbilder' bezeichnet, da sie während der Kompression zur Vorhersage von anderen Frames genutzt werden [114].

B-Frames werden auch als ‚*bi-directionally coded Frames*' bezeichnet, da sie basierend auf dem vorangehenden und nachfolgenden Referenzbild kodiert werden [114]. *Predictively coded Frames* (P-Frames) werden anhand des nächsten vorhergehenden Referenzbildes kodiert, wobei es sich dabei um einen I-Frame oder einen vorhergehenden P-Frame handeln kann [114].

Bestimmte Parameter bestimmen beim Kodierungsprozess, in welcher Reihenfolge I-, P- und B-Frames angeordnet werden. I-Frames sind dabei besonders wichtig, da sie als Referenz genutzt werden und den Anfang einer GOP markieren [114].

Im *ffmpeg*-Kommando ist es wichtig, die *Group Picture Size* mit *-g* anzugeben, da *ffmpeg* ein Videosignal immer nur an einem I-Frame, also dem Anfang einer GOP, segmentieren kann [112]. Um im nächsten Schritt kurze Videosegmente (Chunks) für das HLS zu erzeugen, muss sichergestellt werden, dass *ffmpeg* das Video an den gewünschten Stellen segmentieren kann, die *Group Picture Size* also korrekt definiert ist. Sind zum Beispiel HLS-Chunks mit einer Länge von jeweils zwei Sekunden gewünscht, darf eine GOP bei einer Framerate von 24 fps maximal 48 Bilder enthalten. Ist die *Group Picture Size* zum Beispiel 240, also die GOP zehn Sekunden lang, können die Chunks nicht kleiner als zehn Sekunden lang sein, da *ffmpeg* das Signal nicht innerhalb einer GOP, sondern nur am Anfang segmentieren kann.

Daher wird im Kommando mit *-g 24* definiert, dass jede GOP im Videosignal 24 Bilder enthält. Da das Videosignal eine Framerate von 24 fps besitzt, enthält jede Sekunde Video eine GOP. Dadurch ist es für das zweite Skript möglich, HLS-Chunks mit einer Länge von jeweils einer Sekunde zu generieren (siehe 8.2.2.).

Letztlich wird mit *-f flv* das Ausgangssignal definiert. Da es sich um einen RTMP-Stream handelt, wird das Flash-Videoformat (flv) benötigt [115]. Schließlich muss nur noch der Endpunkt des RTMP-Servers angegeben werden, an den der Stream gesendet wird. Da der Webserver hier lokal läuft, handelt es sich um die *localhost*-Adresse *rtmp://localhost:1935/live/stream1*. Ansonsten wird hier die IP-Adresse des Servers angegeben. *live* ist in diesem Beispiel der Name der Applikation, der in der *nginx.conf*-Datei angegeben wurde, und *stream1* der Streamingschlüssel. Der Streamingschlüssel identifiziert genau einen Stream. Dadurch ist es möglich, einzelne Streams zu unterscheiden, was wichtig ist, wenn mehrere Streams an denselben RTMP-Server geschickt werden.

Sobald das Skript aufgerufen beziehungsweise das Kommando abgesetzt wird, startet die Aufnahme und der Stream wird an den RTMP-Server gesendet. Voraussetzung ist, dass der RTMP-Server läuft und auf die angegebene Portnummer hört.

Eine Alternative zu *UScreenCapture* ist *gdigrab*. Dies ist ein Aufnahmetool, das standardmäßig unter Windows verfügbar ist und wie *UScreenCapture* als Eingabesignal in einem *ffmpeg*-Kommando angegeben werden kann. Das alternative Kommando zur Bildschirmaufnahme mit *gdigrab* sieht wie folgt aus:

```
ffmpeg -f gdigrab -framerate 24 -i desktop -preset fast -tune zerolatency
-keyint_min 24 -sc_threshold 0 -r 24 -g 24 -f flv rtmp://localhost/live/stream1
```

Abbildung 44: ffmpeg-Kommando mit gdigrab, eigener Screenshot

Da *gdigrab* kein directshow-Gerät ist, wird es nicht mit *-f dshow*, sondern mit *-f gdigrab* als Eingabe festgelegt. *-framerate* ermöglicht es, die Framerate der Aufnahme festzulegen, hier auf 24 Frames pro Sekunde. Mit *-i desktop* wird festgelegt, dass der gesamte Desktop aufgenommen wird. Allerdings bietet *gdigrab* auch weitere Optionen, wie *-offset_x*, *-offset_y* und *-video_size*, an, die es ermöglichen, den Aufnahmebereich zu definieren, falls nur ein bestimmter Bildschirmbereich aufgenommen werden soll. Die restlichen Optionen im Kommando bleiben so, wie sie bereits im ersten Kommando mit *UScreenCapture* definiert wurden.

8.2.2. Skript 2: Zerlegung des RTMP-Streams

Das zweite Skript nutzt ebenfalls *ffmpeg* und läuft auf dem *nginx*-Webserver, um den RTMP-Stream in HLS-Segmente zu konvertieren und auf dem Server abzulegen, sodass aufrufende Clients diese herunterladen und abspielen können.

```
ffmpeg -i rtmp://localhost/live/stream1 -preset fast -tune zerolatency
-keyint_min 12 -sc_threshold 0 -r 24 -g 24 -f hls -hls_time 1
-hls_flags delete_segments -hls_list_size 120 /hls/stream1_.m3u8
```

Abbildung 45: Skript 2 zur Zerlegung des RTMP-Streams, eigener Screenshot

Im Gegensatz zum ersten Skript wird der RTMP-Stream, der an den Server geschickt wird, hier als Input (*-i*) definiert. Dabei handelt es sich um dieselbe Adresse, an die das zuvor beschriebene Skript (siehe 8.2.1) die Bildschirmaufnahme sendet. Die Optionen *-preset*, *-tune*, *-keyint_min*, *-sc_threshold* sowie die Framerate und Group Picture Size bleiben gleich. Anders als im ersten Skript wird mit *-f hls* definiert, dass das Eingangssignal von *ffmpeg* in HLS-Segmente zerlegt werden soll. *-hls_time* legt die Dauer jedes Segments fest. Hier ist die Segmentlänge mit einer Sekunde definiert, um die Latenz möglichst gering zu halten. An dieser Stelle ist wichtig, zu erwähnen, dass die Segmente nur eine Sekunde lang sein können, wenn die Group Picture Size maximal eine Sekunde lang ist, also in diesem Fall 24 Bilder in jeder GOP vorhanden sind. Mit der Option *-g 24* wird sichergestellt, dass alle 24 Bilder ein I-Frame gesetzt wird und der RTMP-Stream jede Sekunde in ein HLS-Segment zerlegt werden kann (siehe 8.2.1).

Das HLS-Flag *delete_segments* gibt an, dass alte HLS-Segmente gelöscht werden sollen, sobald die Playlistlänge erreicht ist, die mit *-hls_list_size* definiert ist. Diese ist hier auf 120 begrenzt, was bei einer Segmentlänge von einer Sekunde 120 Sekunden Video entspricht. Das bedeutet, dass jeweils zwei Minuten des Videostreams auf dem Server gespeichert werden, bevor die alten Segmente gelöscht werden. Letztlich erwartet *ffmpeg* den Pfad zur Ausgangsdatei. In diesem Fall ist dies die HLS-Playlist

mit dem Namen *stream1_.m3u8*, die beim Start zusammen mit den Segmenten im Ordner *hls* erstellt wird. Sie enthält Informationen über die abgelegten Segmente und wird von den Clients heruntergeladen, die daraufhin einzelne Segmente anfordern können (siehe 5.4.3.).

Dabei handelt es sich um eine *.m3u8*-Datei. Die *.m3u8*-Dateiendung definiert eine Playlist-Datei, die alle verfügbaren Segmente und die Segmentlänge definiert. Die Segmente werden mithilfe von Segmentnummern benannt, sodass sie in korrekter Reihenfolge abgespielt werden können. Folgende Abbildung zeigt einen Ausschnitt aus einer *.m3u8*-Playlistdatei:

```
#EXTM3U
#EXT-X-VERSION:3
#EXT-X-TARGETDURATION:1
#EXT-X-MEDIA-SEQUENCE:396
#EXTINF:1.000000,
stream1_396.ts
#EXTINF:1.000000,
stream1_397.ts
#EXTINF:1.000000,
stream1_398.ts
```

Abbildung 46: HLS-Playlist, eigener Screenshot

Diese Datei kann in einem späteren Schritt von einem Client abgerufen werden, um Einsicht in die verfügbaren Segmente zu bekommen. Der Client erhält Auskunft über die Segmentlänge (TARGETDURATION) sowie die URLs zu den Segmenten, die hier im selben Ordner wie die Playlist abgelegt sind. Dadurch kann der Client die Chunks herunterladen, nach dem HLS-Prinzip puffern und schließlich wiedergeben (siehe 5.4.3.). Für die Kodierung nutzt *ffmpeg* den H.264-Codec. Die Daten werden dann in *.ts*-Containern gespeichert und abgelegt. In diesem Fall werden alle Segmente lediglich einmal abgelegt, es wird also zunächst auf das adaptive Bitrate-Streaming verzichtet, um den Kodierungsaufwand und damit die Latenz minimal zu halten.

8.2.3. HTML-Seite mit video.js-Videoplayer

Zu Testzwecken wird nun eine HTML-Seite auf dem *nginx*-Webserver im Ordner *src* angelegt, die einen *video.js*-Videoplayer beinhaltet und die HLS-Chunks abspielen kann. Der HTML-Code wurde von der *video.js*-Dokumentation [116] kopiert und es wurden lediglich die Dateipfade zu den entsprechenden CSS- und JavaScript-Dateien angepasst. Außerdem wurden die notwendigen *video.js* JavaScript- und CSS-Dateien von Github [117] heruntergeladen und in demselben Ordner neben die HTML-Datei abgelegt. Folgende Abbildung zeigt den HTML-Code mit Videoplayer:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>HLS Live Streaming</title>
    <link href="video-js.css" rel="stylesheet" />
</head>
<body>
    <video
        id="my-video"
        class="video-js vjs-big-play-centered"
        controls
        muted
        preload="auto"
        width="1920"
        height="1080"
        data-setup="{}">
      <source src="./hls/stream1_.m3u8" type="application/vnd.apple.mpegurl" />
    </video>
    <script src="video.js"></script>
</body>
</html>
```

Abbildung 47: HTML-Seite mit video.js-Videoplayer, eigner Screenshot

Neben der Anpassung der Dateipfade wurde die von *ffmpeg* generierte HLS-Playliste, die auf dem Server im Ordner *hls* neben den HLS-Chunks abgelegt ist, dem *src*-Attribut des Videos zugewiesen. Als *type* wird „application/vnc.apple.mpegurl“ angegeben, sodass der Browser weiß, dass es sich um einzelne HLS-Segmente handelt. Wird die Seite aufgerufen, ist der leere Videoplayer zu sehen. Erst sobald das Streaming gestartet wird und Segmente verfügbar sind, wird der Videostream beim Klick auf den Play-Button wiedergegeben.

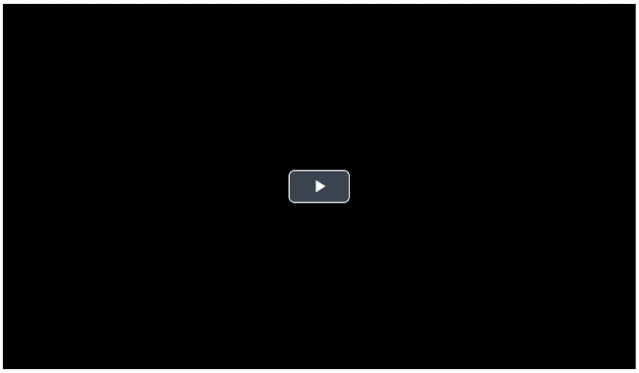

Abbildung 48: leerer Video.js-Player, eigener Screenshot

9. Implementierung Testsystem

Im nächsten Schritt wird die Lösung auf einem Testsystem getestet. Dazu wird die geteilte Systemarchitektur (siehe Abbildung 33) getestet, indem ein entfernter Testrechner als TBM-Rechner fungiert und der lokale Windows-Rechner, der für die bisherige Implementierung genutzt wurde, den Webserver und den Client repräsentiert. Dazu muss im ersten Schritt eine VPN-Verbindung zum entfernten Testrechner aufgebaut werden. Im nächsten Schritt wird innerhalb der Linux-VM auf diesem Rechner ein *nginx*-Server mit RTMP-Modul installiert. Außerdem wird das erste Skript zur Bildschirmaufnahme (siehe 8.2.1.) auf den Testrechner übertragen. Das zweite Skript (siehe 8.2.2.) soll weiterhin auf dem lokalen Rechner laufen und über die VPN-Verbindung den RTMP-Stream vom RTMP-Server abholen und segmentieren. Der HTTP-Webserver zusammen mit dem gehosteten Videoplayer bleibt auf dem lokalen Rechner.

Mit diesem Testaufbau soll getestet werden, wie die in Kapitel 8. beschriebene Implementierung funktioniert, sobald sie auf mehrere Rechner verteilt wird, wie es in der Zukunft sein soll, sobald die Software auf den TBM-Rechnern installiert wird.

9.1. Nginx-RTMP-Server installieren

Im ersten Schritt wird ein *nginx*-Server mit RTMP-Modul auf der Linux-VM installiert, die sich auf dem Testsystem befindet. Um den Server im Hintergrund innerhalb eines Containers laufen zu lassen, wird die Docker-Technologie genutzt. Daher wird im Folgenden zunächst ein kurzer Überblick über Docker gegeben. Danach wird beschrieben, wie ein Dockerfile erstellt wird, um den RTMP-Server als Container auf der Linux-VM zum Laufen zu bringen. Letztlich geht es darum, das erstellte Docker-Image vom lokalen Rechner auf den Testrechner zu übertragen.

9.1.1. Docker-Grundlagen

Docker ist eine open-source-Technologie, die es ermöglicht, Applikationen mit allen notwendigen Abhängigkeiten so zu verpacken, dass sie überall ausgeführt werden können [118]. Die Pakete, in denen Applikationen laufen, werden als ‚Container' bezeichnet und in einer Container-Umgebung ausgeführt [118].

Zwei wichtige Prinzipen von Docker sind Images und Container. Images können mithilfe vordefinierter Vorlagen oder über ein sogenanntes ‚*Dockerfile*' erstellt werden. Ein Dockerfile ist eine Datei, die Anweisungen enthält, die beim Bauen des Images durchlaufen werden [118]. Dadurch kann der Bauprozess automatisiert werden. Bei den Anweisungen innerhalb eines Dockerfiles kann es sich zum Beispiel um das Hinzufügen von Dateien, die Installation von Software-Paketen oder das Absetzen von Kommandos handeln. Sobald ein Image ausgeführt wird, wird von einem Container gesprochen [119].

Ein Container ist eine standardisierte Softwareeinheit, die alle notwendigen Software-Komponenten enthält, um die darin enthaltene Applikation überall ausführen zu können [118]. Das bedeutet, dass ein Container unabhängig von jeglicher Infrastruktur ausgeführt wird und demnach stets gleich funktioniert. Folgende Abbildung zeigt die Docker-Architektur:

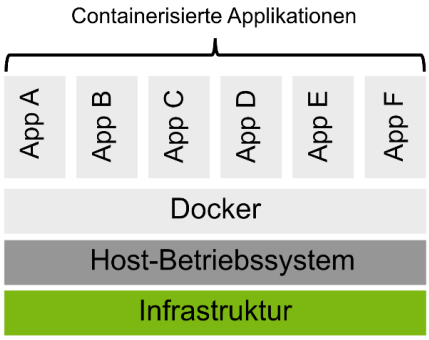

Abbildung 49: Docker-Architektur, eigene Darstellung in Anlehnung an [119]

Wie die Abbildung zeigt, können mehrere Applikationen beziehungsweise Container auf einem Rechner laufen und sich das darunterliegende Betriebssystem teilen. Jeder Container ist dabei eine isolierte Einheit, die in der Docker-Engine läuft [119].

9.1.2. Dockerfile erstellen

Für die Installation des *nginx*-Webservers in der Linux-VM wurde zunächst ein Dockerfile geschrieben, aus dem das Docker-Image erstellt wird. Das Dockerfile wird zunächst in einem Windows-Subsystem für Linux (WSL) erstellt, da es im nächsten Schritt als Docker-Image in die Linux-VM auf dem Testrechner übertragen werden soll, um dort als Container ausgeführt zu werden. Folgende Abbildung zeigt das geschriebene Dockerfile:

```
FROM tiangolo/nginx-rtmp:latest

RUN apt-get update

RUN apt install -y python3-pip
RUN pip3 install supervisor

COPY nginx.conf /etc/nginx/nginx.conf

ADD supervisord.conf /usr/local/etc/supervisord.conf
CMD ["supervisord"]
```

Abbildung 50: Dockerfile nginx-RTMP-Server, eigener Screenshot

Docker-Images können auf bereits existierenden Images bestehen. So wird hier an erster Stelle mit *FROM* definiert, auf welchem Image das neue Image basieren soll. In diesem Fall ist es das Image *nginx*-

rtmp von *tiangolo*, das auf Dockerhub zu finden ist [120]. Dies ist das Basis-Image, das hier genutzt wird.

Im nächsten Schritt wird mit *apt-get update* ein Update durchgeführt, um die neuesten Pakete zur Verfügung zu stellen. Danach wird der Paketmanager *pip* installiert, um im Anschluss *supervisor* zu installieren. *supervisor* ist ein System, das es ermöglicht, mehrere UNIX-Prozesse zu verwalten und zu kontrollieren [121].

Mit *COPY* wird die Konfigurationsdatei des *nginx*-Webservers vom lokalen System in das Image kopiert. Dadurch wird sichergestellt, dass jeder Webserver gleich konfiguriert ist, auch wenn dasselbe Docker-Image später als Container auf mehreren Rechnern läuft. Die Konfiguration ist dabei dieselbe, die bereits für den RTMP-Server auf dem Windows-Rechner eingesetzt wurde (siehe 8.1.3.).

ADD fügt die *supervisord.conf*-Datei in den entsprechenden Ordner hinzu. Darin können die UNIX-Prozesse definiert werden, die mit *CMD["supervisord"]* gestartet werden. In diesem Fall handelt es sich lediglich um den *nginx*-Webserver, allerdings kann es vorkommen, dass ein Docker-Image mehrere Prozesse gleichzeitig starten soll, wozu *supervisor* notwendig ist.

Im nächsten Schritt kann das Docker-Image erstellt werden. Dazu wird folgendes Kommando im Ordner abgesetzt, in dem sich das Dockerfile befindet:

```
docker build -t image-name .
```

Abbildung 51: Docker-Image bauen, eigener Screenshot

Ist das Docker-Image erfolgreich gebaut, erscheint der Image-Name (hier: *image-name*) in der Liste, die ausgegeben wird, sobald das Kommando *docker images* abgesetzt wird. Das Image kann im nächsten Schritt in die Linux-VM übertragen werden, um dort gestartet zu werden. Im nächsten Abschnitt wird dieser Schritt behandelt.

9.1.3. Übertragung des Dockerfiles und Starten des Containers

Da das Docker-Image nicht in der lokalen WSL-Umgebung, sondern in der Linux-VM auf dem Testrechner gestartet werden soll, muss das Docker-Image zunächst in die Linux-VM übertragen werden. Dazu wird das Image erst in eine *.tar*-Datei konvertiert. Dies geschieht über folgendes Kommando:

```
docker save -o /filepath/outputfile.tar image-name
```

Abbildung 52: Docker-Image speichern, eigener Screenshot

outputfile.tar ist dabei die generierte *.tar*-Datei, die das Image enthält, und *image-name* der Name des Images, das gespeichert werden soll.

Nun ist das Image in einem Format gespeichert, das zwischen Rechnern ausgetauscht werden kann. Im nächsten Schritt muss die Datei auf den Windows-Rechner übertragen werden, auf dem die WSL-Umgebung läuft. Dazu wird die Datei mit *cp outputfile.tar filepath* an die entsprechende Stelle kopiert, wobei *filepath* dem Dateipfad auf dem Windows-Rechner entspricht, wo die Datei gespeichert werden soll.

Als nächstes muss eine Verbindung zum Testsystem aufgebaut werden, um das Docker-Image auf die Linux-VM zu übertragen. Dazu ist eine SSH-Verbindung notwendig. SSH steht für ‚Secure Shell' und ist ein Tool, das es ermöglicht, auf sichere Weise Systeme zu administrieren beziehungsweise Dateien zu übertragen [122]. Das SSH-Protokoll verschlüsselt die Verbindung zwischen einem Client, in diesem Fall dem lokalen Windows-Rechner, und einem Server, hier der Linux-VM. Dadurch wird unter anderem die Datenübertragung vor Netzwerkangriffen geschützt [122].

Folgende Abbildung zeigt den Aufbau von SSH-Verbindungen zwischen einem Client und einem Server:

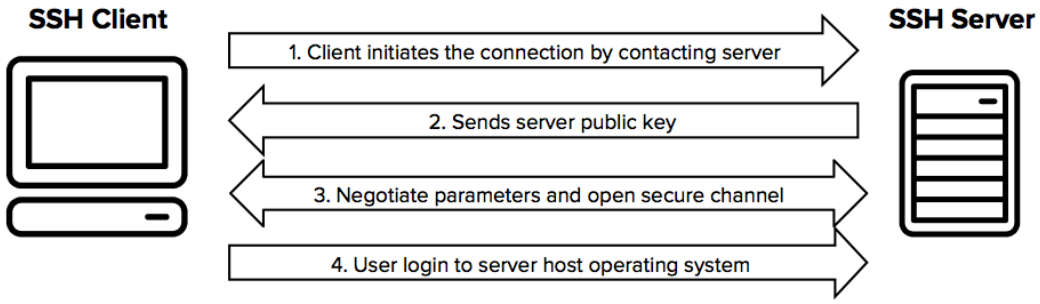

Abbildung 53: SSH-Protokoll, Quelle: [122]

Dabei sendet der Client zunächst eine Verbindungsanfrage an den Server, der mit seinem öffentlichen Schlüssel antwortet [122]. Im nächsten Schritt werden Verbindungsparameter verhandelt, bevor sich der Nutzer schließlich einloggen kann [122].

Um eine Verbindung zwischen dem lokalen Windows-Rechner und der Linux-VM auf dem Testsystem aufzubauen, muss zunächst eine VPN-Verbindung zwischen den Rechnern aufgebaut werden. Im Anschluss kann folgendes Kommando in einem Windows-Konsolenfenster abgesetzt werden:

```
ssh user@IP-ADRESSE -p XYZ
```

Abbildung 54: ssh-Verbindungsaufbau

Die IP-Adresse ist hier die Adresse der Linux-VM und *XYZ* die Portnummer. Um das erstellte Docker-Image vom Windows-Rechner auf die Linux-VM zu übertragen, wird folgendes Kommando abgesetzt:

```
scp -P XYZ outputfile.tar user@IP-ADRESSE:./outputfile.tar
```

Abbildung 55: Dateitransfer auf Linux-VM, eigener Screenshot

Ist die Dateiübertragung abgeschlossen, kann das Image in der Linux-VM geladen werden, um im Anschluss als Container ausgeführt zu werden. Dazu wird in der Linux-VM folgendes Kommando abgesetzt:

```
docker load < outputfile.tar
```

Abbildung 56: Docker-Image laden, eigener Screenshot

Nun ist das Docker-Image in der Liste der verfügbaren Docker-Images verfügbar, die beim Absetzen des Kommandos *docker images* ausgegeben wird, und kann mit folgendem Kommando gestartet werden:

```
docker run -p portNummer:containerPortnummer image-name
```

Abbildung 57: Docker-Image starten, eigener Screenshot

Mit *-p* kann dabei angegeben werden, unter welcher Portnummer der *nginx*-Webserver außerhalb des Containers erreichbar sein soll. *containerPortnummer* entspricht dabei der Portnummer innerhalb des Containers und *portNummer* der Nummer außerhalb des Containers. Dieses sogenannte ‚Mapping' ist wichtig, damit der RTMP-Server auch außerhalb des Containers erreichbar ist und der RTMP-Stream durch *ffmpeg* an diesen gesendet werden kann. *image-name* entspricht dem Namen des erstellten Docker-Images. Nun ist der RTMP-Server auf dem Testrechner in der Linux-VM installiert und läuft. Mit *docker ps* lassen sich alle laufenden Container ansehen.

9.2. Skript zum automatischen Start des Streamings

Im nächsten Schritt ging es darum, ein Skript zu schreiben, das einmalig auf die TBM-Rechner beziehungsweise an dieser Stelle den Windows-Testrechner übertragen und gestartet wird, um das Streaming zu starten. Das Ziel ist es, dass Kunden das Skript ohne viel Aufwand einmalig auf ihrem TBM-Rechner starten können, um einen Stream des Bildschirminhaltes auf *CONNECTED* zu sehen.

Beim Skript handelt es sich um eine Batch-Datei, also eine Datei, die mehrere Kommandos hintereinander ausführt. Neben dieser Datei, die im Folgenden ausführlich beschrieben wird, müssen die *ffmpeg*-Dateien sowie die Installationsdatei für *UScreenCapture* auf den Testrechner übertragen werden.

Im ersten Schritt geht es darum, die Bildschirmgröße des Rechners zu ermitteln, um die Videogröße bei der Bildschirmaufnahme festzulegen und gegebenenfalls den Aufnahmebereich mit einem Offset in x- und y-Richtung einzugrenzen. Dies könnte zum Einsatz kommen, wenn die TBM-Rechner über mehrere Monitore verfügen, aber nur ein bestimmter Bereich für das Streaming auf *CONNECTED* bereitgestellt werden soll.

Das Kommando, um die Bildschirmgröße des Desktops eines Windows-Rechners zu ermitteln, ist das folgende:

```
wmic desktopmonitor get screenheight, screenwidth

ScreenHeight  ScreenWidth
1080          1920
```

Abbildung 58: Ermittlung Desktop-Größe, eigener Screenshot

Das Kommando gibt die Bildschirmbreite und die Bildschirmhöhe als Zahlenwerte aus. Um die Zahlenwerte auszulesen, wurde das Kommando wie folgt in das Skript integriert:

```
for /f "tokens=1,2 delims==" %%i in ('wmic desktopmonitor
get screenheight^,screenwidth /value ^| find "="') do (
  if "%%i"=="ScreenHeight" set height=%%j
  if "%%i"=="ScreenWidth" set width=%%j
)
echo your screen is %width% * %height% pixels
```

Abbildung 59: Ermittlung Desktop-Höhe & -Breite, eigener Screenshot

Um die Bildschirmhöhe und -breite in Variablen zu speichern, wird eine for-Schleife durchlaufen, die die Ausgabe des zuvor beschriebenen Kommandos durchsucht. Mithilfe von if-Bedingungen wird nach den Strings „ScreenHeight“ beziehungsweise „ScreenWidth“ gesucht und bei einem Treffer die Ausgabe in den Variablen *height* und *width* gespeichert. Mit echo werden die ermittelten Zahlenwerte auf der Konsole ausgegeben.

Nachdem die Bildschirmgröße ermittelt wurde, kann das Streaming gestartet werden. Dazu können zwei verschiedene Tools zum Einsatz kommen: entweder *gdigrab*, das standardmäßig auf Windows-Rechnern verfügbar ist, oder *UScreenCapture*, das erst noch installiert werden muss (siehe 8.1.2.).

Falls *gdigrab* eingesetzt werden soll, kann das Streaming direkt gestartet werden:

```
ffmpeg -f gdigrab -framerate 30 -offset_x 0 -offset_y 0 -video_size %width%x%height%
-i desktop -preset fast -tune zerolatency -keyint_min 24 -sc_threshold 0 -r 24 -g 24
-f flv rtmp://IP-Adresse:1935/live/stream1
```

Abbildung 60: ffmpeg-Streaming mit gdigrab, eigener Screenshot

Da dieses Kommando bereits ausführlich in Kapitel 8.2.1. erläutert wurde, wird an dieser Stelle nicht weiter auf die eingesetzten *ffmpeg*-Optionen eingegangen. Der einzige Unterschied besteht hier darin, dass die Variablen *height* und *width* genutzt werden, um die Videogröße anzugeben. Der Offset in x- und y-Richtung ist hier mit 0 definiert, allerdings kann dieser variieren, falls nur ein bestimmter Bildschirmbereich aufgenommen werden soll. Wichtig ist außerdem, dass die IP-Adresse und die

Portnummer des RTMP-Streams an die Konfiguration des RTMP-Servers auf der Linux-VM angepasst werden, damit der Stream an diesen gesendet werden kann.

Nachdem das Skript dieses Kommando absetzt, beginnt das Streaming. Dabei wird der Stream im Flash-Format als RTMP-Stream an den zuvor installierten *nginx*-Server in der Linux-VM gesendet und kann vom zweiten Skript auf dem lokalen Windowsrechner segmentiert werden.

Neben *gdigrab* kann *UScreenCapture* als Alternative eingesetzt werden. Allerdings muss dieses Tool gegebenenfalls erst installiert werden (siehe 8.1.2.). Soll *UScreenCapture* anstatt *gdigrab* eingesetzt werden, wird das zuvor genannte *ffmpeg*-Kommando (siehe Abbildung 60) auskommentiert und stattdessen werden folgende Kommandos im Skript durchlaufen:

```
echo checking if UScreenCapture is already installed ...
ffmpeg -list_devices true -f dshow -i dummy 2>&1 | findstr "UScreenCapture" >temp.txt

del temp.txt
IF errorlevel 1 ( echo Installling UScreenCapture... && cd .. &&
                  cd UScreenCapturex642018 && UScreenCapture_x64.msi && echo Finished!)^
ELSE (echo UScreenCapture already installed!)
```

Abbildung 61: UScreenCapture-Installation, eigener Screenshot

Im ersten Schritt wird geprüft, ob *UScreenCapture* bereits auf dem Rechner installiert ist oder nicht. Dazu wird mit *ffmpeg -list_devices* eine Liste aller directshow-Geräte, also aller Video- und Audiogeräte, über die der Rechner verfügt, ausgegeben. Die Ausgabe wird schließlich mit dem Pipe-Operator (|) an das Kommando *findstr* weitergeleitet, das diese nach dem String „UScreenCapture“ durchsucht und das Ergebnis in die Textdatei *temp.txt* schreibt. Falls *UScreenCapture* bereits installiert ist, wird das Tool in der Liste aller Geräte als virtuelles Videoeingabegerät gefunden und demnach in die Datei geschrieben, die im nächsten Schritt mit *del temp.txt* gelöscht wird.

Danach wird geprüft, ob das Tool gefunden wurde oder nicht. Dabei wird mit einer if-Abfrage geprüft, ob die Variable *errorlevel* dem Wert 1 entspricht oder nicht. Wurde *UScreenCapture* mit *findstr* nicht gefunden, wird die Variable auf 1 gesetzt. Dies geschieht automatisch, ohne dass die Variable angelegt oder konkret zugewiesen werden muss. In diesem Fall ist *UScreenCapture* noch nicht auf dem Rechner installiert und die Installationsdatei, die sich auch im Ordner befindet, wird aufgerufen. Der Nutzer muss sich dann durch die Installationsanweisungen klicken, bis die Installation abgeschlossen wurde. Ist die Variable *errorlevel* nicht auf 1 gesetzt, ist *UScreenCapture* bereits installiert.

Nun kann das Streaming mit *UScreenCapture* gestartet werden. Dabei wird das angepasste *ffmpeg*-Kommando aufgerufen, das bereits in Abschnitt 8.2.1. erläutert wurde:

```
ffmpeg -f dshow -i video="UScreenCapture" -framerate 30 -rtbufsize 1024M
-vf crop=%width%:%height% -c:v libx264rgb -crf 0 -preset ultrafast
-tune zerolatency -keyint_min 24 -sc_threshold 0
-r 24 -g 24 -f flv rtmp://IP-Adresse:1935/live/stream1
```

Abbildung 62: ffmpeg-Streaming mit UScreenCapture, eigener Screenshot

Wie bei der ersten Option mit *gdigrab* beginnt nun das Streaming und die Aufnahme wird als RTMP-Stream an den RTMP-Server gesendet.

Das vollständige Skript zum automatischen Start des Streamings befindet sich im Anhang (siehe A).

Um den erstellten Ordner mit allen notwendigen Dateien auf das Testsystem zu übertragen, kann eine Verbindung zum Testrechner per TeamViewer, einem Tool für den Fernzugriff, hergestellt werden. Dazu werden lediglich die ID und das Passwort des Rechners benötigt. Sobald die TeamViewer-Verbindung hergestellt wurde, kann die Dateiübertragung geöffnet werden.

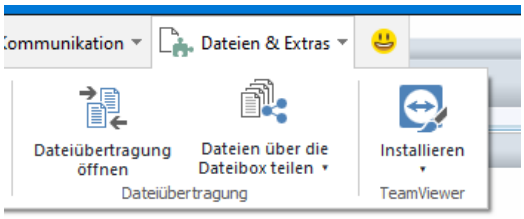

Abbildung 63: Dateiübertragung TeamViewer, eigener Screenshot

Sobald diese geöffnet ist, können die zu übertragenden Dateien, in diesem Fall der Ordner mit den *ffmpeg*-Dateien, *UScreenCapture* und dem Skript zum automatischen Start des Streamings, ausgewählt werden. Auf dem Testsystem wird der entsprechende Speicherort gewählt und danach der Ordner an das Testsystem übertragen. Dieser enthält die folgenden Dateien:

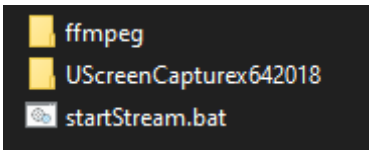

Abbildung 64: Übertragene Dateien, eigener Screenshot

Sobald der Ordner übertragen wurde, kann das Streaming durch Klick auf das Skript *startStream.bat* (siehe Abbildung 64) im Testsystem gestartet werden. Voraussetzung ist, dass der RTMP-Server auf der Linux-VM und das zweite Skript auf dem Webserver läuft.

9.3. Docker-Container erstellen

Nun geht es darum, ein weiteres Docker-Image zu erstellen, das in der Zukunft in der Cloud als Container gestartet werden kann. Dieses Image soll einen *nginx*-Webserver enthalten, auf dem das zweite Skript (siehe 8.2.2.) laufen soll, das den RTMP-Stream segmentiert. Diese Komponenten laufen in diesem Testaufbau noch auf dem lokalen Windows-Rechner, allerdings sollen diese in der Zukunft in der Cloud laufen, um den Videostream in *CONNECTED* zu integrieren.

Daher wird zunächst folgendes Dockerfile erstellt, das in einem späteren Schritt als Container in der Cloud zum Laufen gebracht werden soll:

```
FROM nginx:latest

RUN apt-get update

RUN apt-get install -y ffmpeg
RUN apt install -y python3-venv python3-pip
RUN pip install supervisor

COPY src/html /usr/share/nginx/html

ADD supervisord.conf /usr/local/etc/supervisord.conf
CMD ["supervisord"]
```

Abbildung 65: Docker-Image mit nginx-Webserver, eigener Screenshot

Das erstellte Image basiert auf dem neusten *nginx*-Image (ohne RTMP-Modul) von Dockerhub [123]. Im ersten Schritt wird ein Update durchgeführt, um sicherzustellen, dass die aktuellen Softwarepakete verfügbar sind. Im Anschluss wird *ffmpeg* installiert, was vom Skript zur Segmentierung des RTMP-Streams benötigt wird. Der Paketmanager *pip* wird als nächstes heruntergeladen, um daraufhin *supervisor* zu installieren. Wie bereits beschrieben wurde, wird *supervisor* benötigt, um innerhalb eines Docker-Containers mehrere UNIX-Prozesse zu starten, in diesem Fall den nginx-Webserver und das *ffmpeg*-Kommando.

Danach wird der Ordner mit dem Namen *html* an die entsprechende Stelle im Docker-Image kopiert, damit die darin enthaltene HTML-Seite von *nginx* gehostet werden kann. Letztlich wird die *supervisord.conf*-Konfigurationsdatei in das Image kopiert und *supervisor* mit *CMD ["supervisord"]* gestartet.

Nun soll ein Blick auf die *supervisord.conf*-Datei geworfen werden, da diese dafür zuständig ist, den *nginx*-Webserver und das Streaming mit *ffmpeg* zu starten.

```
[supervisord]
nodaemon=true

[program:nginx]
command=nginx
startsecs=1
startretries=3

[program:ffmpeg]
command=ffmpeg -i rtmp://10.126.6.155:1935/live/stream1
-preset fast -tune zerolatency -sc_threshold 0 -r 24 -g 24
-f hls -hls_time 1 -hls_flags delete_segments -hls_list_size 160
/usr/share/nginx/html/hls/stream1.m3u8
startsecs=1
startretries=3
```

Abbildung 66: supervisord.conf, eigener Screenshot

An erster Stelle wird ein Abschnitt, *[supervisord]*, definiert, der die globalen Einstellungen für den *supervisor*-Prozess enthält. Mit *nodaemon=true* wird definiert, dass *supervisord* im Vordergrund anstatt im Hintergrund gestartet wird [124].

Als nächstes werden zwei Programm-Abschnitte mit *[program:x]* definiert, wobei *x* dem Programmnamen entspricht. Ein solcher Abschnitt teilt *supervisord* mit, welches Programm gestartet werden soll [124]. In diesem Fall werden zwei solche Abschnitte erstellt: einer für den *nginx*-Webserver und einer für *ffmpeg*. Jeder Abschnitt kann verschiedene Optionen enthalten. So wird jeweils an erster Stelle mit *command* definiert, welches Kommando abgesetzt werden soll, um das jeweilige Programm zu starten. Der Webserver wird mit dem einfachen Kommando *nginx* gestartet, während es sich beim zweiten Kommando für *ffmpeg* um das zweite Skript (siehe 8.2.2.) handelt, das den RTMP-Stream in Segmente zerlegt. Mit *startsecs* wird angegeben, wie viele Sekunden das Programm laufen muss, um als erfolgreich laufender Prozess angesehen zu werden. *startretries* gibt an, wie häufig versucht wird, das Programm nach einem fehlerhaften Start neu zu starten [124].

Beim Aufruf von *supervisord*, wie es am Ende des Dockerfiles angegeben wurde, wird diese Konfigurationsdatei durchlaufen und dabei werden die definierten Programme *nginx* und *ffmpeg* gestartet. Mit der Voraussetzung, dass das erste Skript bereits auf dem Testrechner läuft und den RTMP-Stream an den RTMP-Server in der Linux-VM sendet, sollte der Stream bei erfolgreichem Start des Containers durch das zweite *ffmpeg*-Kommando segmentiert werden. Außerdem sollte die HTML-Seite, die auf dem Webserver gehostet wird, im Browser aufgerufen werden können. Der Stream sollte beim Start des Videoplayers abgespielt werden.

9.4. Zusammenfassung Testsystem

Bei der Implementierung des Testsystems, insbesondere bei der Herstellung der Verbindung zwischen dem Container in der Cloud und dem TBM-Testrechner, sind Probleme aufgetreten. Nachdem das Skript zur Bildschirmaufnahme auf dem TBM-Testrechner und der Container zur Zerlegung des Streams in der Cloud zum Laufen gebracht wurden, musste die Verbindung zwischen der Cloud und dem Testrechner hergestellt werden, um den RTMP-Stream überhaupt vom Testrechner in die Cloud zu übertragen. Allerdings ist diese Verbindung fehlgeschlagen.

Aus Sicherheitsgründen können Container in der Cloud nicht direkt auf TBM-Rechner zugreifen. Stattdessen verwaltet ein spezieller Container alle Verbindungen zwischen anderen Containern und den TBM-Rechnern. Dabei handelt es sich um einen HTTP-Proxy, der alle TBM-Netze verwaltet. Da der generierte RTMP-Stream jedoch nicht über HTTP übertragen werden kann [61], konnte keine funktionsfähige Verbindung aufgebaut werden, um den RTMP-Stream vom Testrechner in die Cloud zu übertragen. Das bedeutet, dass der RTMP-Stream vom *ffmpeg*-Kommando im Container nicht entgegengenommen und die Architektur, wie sie in Abbildung 33 dargestellt ist, nicht erfolgreich umgesetzt werden konnte.

Aus diesem Grund musste die Systemarchitektur des Projektes überarbeitet und angepasst werden, um den Transport des Streams von den Rechnern in die Cloud zu ermöglichen. Die neue Architektur und deren Implementierung werden in Kapitel 10 erläutert.

10. Architektur-Erweiterung

Da die ursprüngliche Projekt-Architektur während des Testens nicht mit der gegebenen Infrastruktur vereint werden konnte (siehe 9.), wurde eine erweiterte Architektur entwickelt, um die Verbindung zwischen TBM-Rechnern und dem Container in der Cloud zu schaffen. Die Idee ist, dass die Bildschirmaufnahme direkt auf dem TBM-Rechner anstatt erst in der Cloud in Segmente zerlegt wird und diese auf dessen Festplatte abgelegt werden. Um die einzelnen Segmente in die Cloud zu transportieren, wird eine Rest-API in C# entwickelt. Außerdem wird ein Rest-Client in der Cloud benötigt, der die einzelnen Segmente per HTTP-Get-Request abruft und auf dem Webserver ablegt. Der Container in der Cloud und der TBM-Rechner können dann mit einer regulären VPN-Verbindung kommunizieren, so wie es auch für die ursprüngliche Architektur gedacht war. Da die Videosegmente im Gegensatz zum RTMP-Stream per HTTP übertragen werden können, konnte diese neue Architektur umgesetzt werden. Folgende Abbildung zeigt eine Darstellung der neuen Projektarchitektur:

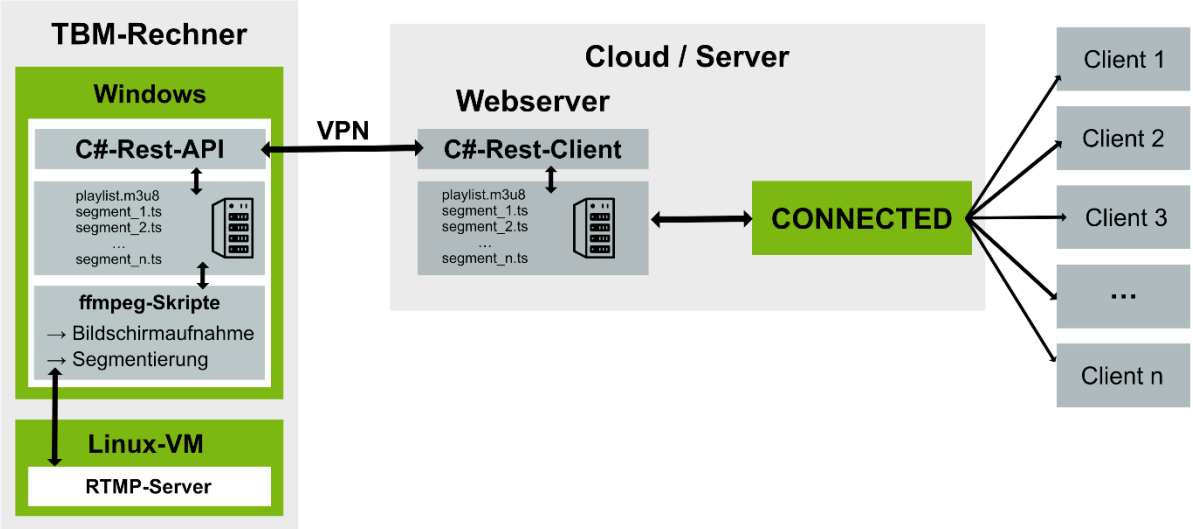

Abbildung 67: Neue Architektur des Prototyps, eigene Darstellung

Im Folgenden geht es um die Entwicklung einer Rest-API sowie eines Rest-Clients in C#. Außerdem wird erläutert, wie diese kommunizieren, um die Streaming-Lösung zum Laufen zu bringen.

10.1. Implementierung C#-Rest-API

Im ersten Schritt ging es darum, eine Rest-API in C# zu entwickeln, die auf den Rechnern auf den Tunnelbohrmaschinen laufen soll und dafür zuständig ist, die generierten HLS-Segmente zur Verfügung zu stellen. Dazu werden mehrere Endpunkte benötigt, die vom C#-Rest-Client abgerufen werden können, um einzelne Videosegmente herunterzuladen. Im ersten Schritt wurde ein neues C#-Projekt angelegt. Der vollständige Code der entwickelten C#-Rest-API befindet sich im Anhang (siehe Anhang B.). Im Folgenden werden einige wichtige Codeausschnitte detaillierter betrachtet.

10.1.1. API-Endpunkte

Innerhalb der API wurden mehrere Endpunkte geschrieben. Dabei soll es möglich sein, die vom *ffmpeg*-Kommando erstellte HLS-Playlist sowie einzelne Videosegmente abrufen zu können.

Folgende Abbildung zeigt den Endpunkt, womit die Playlist-Datei abgerufen werden kann:

```
// GET: api/segment/playlist
[HttpGet("playlist")]
0 references
public async Task GetPlaylist()
{
    Console.WriteLine("Called /api/segment/playlist");
    string playlistFilePath = filePath + "stream1_.m3u8";
    string playlistTempFilePath = filePath + "stream1_temp.m3u8";

    try{
        System.IO.File.Copy(playlistFilePath, playlistTempFilePath, true);
        using(FileStream stream1 = new FileStream(playlistTempFilePath, FileMode.Open, FileAcc
        {
            Response.StatusCode = (int)HttpStatusCode.OK;
            Response.Headers.Add( HeaderNames.ContentDisposition, $"attachment; filename=\"{Gu
            Response.Headers.Add( HeaderNames.ContentType, "application/vnd.apple.mpegurl"  );
            await stream1.CopyToAsync(Response.Body);
            await Response.Body.FlushAsync();
        }
        System.IO.File.Delete(playlistTempFilePath);
```

Abbildung 68: C#-Rest-Endpunkt (Playlist), eigener Screenshot

Zunächst wird mit *HttpGet* der Endpunkt definiert, über den die Playlist abgerufen werden kann. In diesem Fall lautet der Endpunkt „*playlist*“ beziehungsweise */api/segment/playlist*, da die Route für die komplette API bereits zuvor mit */api/segment/* definiert wurde (siehe Anhang B).

Danach wird die asynchrone Methode *GetPlaylist()* definiert. Darin wird zunächst eine Konsolenausgabe durchgeführt und der Dateipfad zur Playlist-Datei auf der Festplatte mit dem Namen *stream1_.m3u8* definiert. Außerdem wird ein weiterer Pfad zu einer temporären Playlist-Datei mit dem Namen *stream1_temp.m3u8* als Zeichenkette angelegt. Im Anschluss wird der Dateiinhalt der ursprünglichen Datei zum entsprechenden Zeitpunkt mit *File.Copy()* in die temporäre Datei kopiert.

File.Copy() ist ein atomarer Dateivorgang [125], was bedeutet, dass die temporäre Datei entweder den vorherigen Stand der Playlist-Datei oder den neuen Stand nach dem Kopiervorgang beinhaltet. Da die Playlist-Datei kontinuierlich von *ffmpeg* geöffnet und mit neuen Informationen überschrieben wird, wird verhindert, dass versucht wird, die Datei zu öffnen, während sie von *ffmpeg* bearbeitet wird. Die temporäre Datei stellt dementsprechend sicher, dass dies nicht passiert.

Im nächsten Schritt wird die temporäre Datei als *FileStream* geöffnet und eine HTTP-Antwort erstellt, die als Inhalt die Playlist-Datei vom Typ „*application/vnd.apple.mpegurl*“ zurückgibt. Beim Aufruf von *https://localhost:port/api/segment/playlist* wird dementsprechend der aktuelle Stand der Playlist-Datei heruntergeladen. Dabei ist *port* die definierte Portnummer, auf die die API hört. Die Playlist-Datei wird vom C#-Rest-Client benötigt, um zu Beginn die erste Segmentnummer zu ermitteln (siehe 10.2.).

Im nächsten Schritt wurde ein weiterer API-Endpunkt entwickelt, der beim Aufruf ein Videosegment mit einer bestimmten ID beziehungsweise Sequenznummer herunterlädt. Folgende Abbildung zeigt den C#-Code des Endpunkts:

```
// GET: api/segment/{id} -->get specific .ts File (hls-segment)
[HttpGet("{id}")]
0 references
public async Task  GetHLS_segment(int id)
{
    Console.WriteLine("Called /api/segment/" + id);
    string hlsSegmentPath = filePath + "stream1_" + id + ".ts";
    try{
        using(FileStream stream = new FileStream(hlsSegmentPath, FileMode.Open, FileAcces
            Response.StatusCode = (int)HttpStatusCode.OK;
            Response.Headers.Add( HeaderNames.ContentDisposition, $"attachment; filename=
            Response.Headers.Add( HeaderNames.ContentType, "application/octet-stream"  );
            await stream.CopyToAsync(Response.Body);
            await Response.Body.FlushAsync();
            stream.Close();
        }
    }
```

Abbildung 69: C#-Rest-Endpunkt (ein Segment), eigener Screenshot

Die Route zum Endpunkt wurde als */api/segment/id* definiert, wobei *id* der Sequenznummer des angeforderten Segments entspricht. Welches Segment angefordert wird, wird vom Client bestimmt (siehe 10.2.).

Der Pfad zum angeforderten Segment wird als Zeichenkette (*string*) definiert. Da jedes Segment im Format *stream1_{id}.ts* benannt ist, kann durch Einsetzen der übergebenen ID der Dateiname bestimmt werden. Danach wird die Segment-Datei als *FileStream* geöffnet und eine HTTP-Antwort erstellt. Dabei werden der Statuscode, der eigentliche Inhalt und der Typ des Inhaltes definiert. Zurückgegeben wird schließlich die erstellte *Response* mit dem darin enthaltenen Stream. Beim Aufruf des Endpunktes wird das jeweilige Segment heruntergeladen.

10.1.2. API-Konfiguration

Im nächsten Schritt musste die Datei *appsettings.json* angepasst werden, um sicherzustellen, dass die API auch unter anderen IP-Adressen und nicht nur der *localhost*-Adresse erreichbar ist. Zudem ist dies notwendig, um dieselbe API auf mehreren Rechnern laufen zu lassen, sodass die Lösung in der Zukunft für alle TBM-Rechner zum Einsatz kommen kann, ohne für jeden Rechner individuell angepasst werden zu müssen.

Abbildung 70: Erweiterung appsettings.json, eigener Screenshot

Die *appsettings.json* wird um einen „Kestrel"-Endpunkt erweitert. *Kestrel* ist ein Webserver für ASP.NET Core und unterstützt unter anderem HTTPS und HTTP [126]. Hier wird ein HTTP-Endpunkt angegeben, auf den die API hört. Da die Verbindung zwischen Client und API per VPN verschlüsselt wird, genügt an dieser Stelle HTTP, und es wurde auf HTTPS verzichtet. Als *Url* wird *http://*:5000* angegeben. Das bedeutet, dass die API auf allen IP-Adressen des Rechners und der Portnummer 5000 erreichbar ist. Damit kann die entwickelte C#-Rest-API auf allen TBM-Rechnern eingesetzt werden, ohne den Code für jeden Rechner anpassen zu müssen.

10.1.3. Erstellung einer ausführbaren .exe

Um zu verhindern, dass auf jedem TBM-Rechner .NET installiert werden muss, um die API auszuführen, ging es im nächsten Schritt darum, das C#-Projekt als einzelne ausführbare Datei zu exportieren. Diese Datei soll alle notwendigen Abhängigkeiten enthalten und ohne weitere Installationen oder Konfigurationen auf dem TBM-Rechner ausgeführt werden können, um die API zu starten. Dazu wurden im ersten Schritt die Projekteinstellungen erweitert:

```
<PublishSingleFile>true</PublishSingleFile>
<SelfContained>true</SelfContained>
<RuntimeIdentifier>win-x64</RuntimeIdentifier>
<PublishReadyToRun>true</PublishReadyToRun>
```

Abbildung 71: Erweiterung .csproj-Datei, eigener Screenshot

Um sicherzustellen, dass beim Export lediglich eine Datei generiert wird, die unabhängig von diversen Projektdateien und ohne .NET-Installation ausgeführt werden kann, wurden einige Tags in der .csproj-Datei des Projektes ergänzt. *PublishSingleFile* gibt an, dass nur eine Datei erstellt wird [127]. *SelfContained* definiert, dass die Datei in sich geschlossen ist und nicht abhängig vom eingesetzten Framework ist [127]. *RunTimeIdentifier* gibt das Betriebssystem an, auf dem die Datei ausgeführt werden soll, und *PublishReadyToRun* definiert, dass die .exe-Datei bereit ist, ausgeführt zu werden [128].

Um die .exe-Datei zu generieren, kann in der Kommandozeile im entsprechenden Projektordner folgendes Kommando abgesetzt werden:

```
dotnet publish -r win-x64 -c winFinal -p:PublishSingleFile=true --self-contained true
```

Abbildung 72: Erstellung .exe, eigener Screenshot

Mit *dotnet publish* wird die Datei veröffentlicht. *-r* gibt das Ziel-Betriebssystem an und *-c* den Zielordner, in den die Datei abgelegt werden soll. Weiterhin wird definiert, dass eine einzelne Datei veröffentlicht werden soll, die in sich geschlossen und ohne .NET-Installation läuft.

Schließlich werden eine .exe-Datei und die *appsettings.json*-Datei erhalten, die neben der erstgenannten abgelegt ist. Diese wird nicht in die .exe-Datei integriert, sodass Konfigurationen, wie das Anpassen der Portnummer, auch nach dem Export noch möglich sind. Diese Dateien können nun mit den *ffmpeg*-Skripten auf einen Testrechner beziehungsweise einen TBM-Rechner übertragen werden, um dort gestartet zu werden.

10.2. Implementierung C#-Rest-Client

Nachdem die C#-Rest-API fertiggestellt wurde, ging es darum, einen Rest-Client in C# zu schreiben, der in regelmäßigen Abständen die HLS-Videosegmente von der Rest-API herunterlädt und die HLS-Playliste erzeugt, die von Browsern heruntergeladen wird, um die Segmente als Videostream abzuspielen. Dazu wurde zunächst eine Konsolenapplikation in Visual Studio erstellt und die generierte Klasse Program.cs erweitert. Der gesamte C#-Code befindet sich im Anhang C, allerdings werden die wichtigsten Ausschnitte im Folgenden detaillierter betrachtet.

Im ersten Schritt ist der C#-Rest-Client dafür verantwortlich, die Playliste von der API herunterzuladen. Daraus kann der Client ermitteln, welches Segment zuletzt generiert wurde, also welche Segmentnummer die aktuellste ist. Nachdem diese Segmentnummer bestimmt wurde, wird ein Timer gestartet, der dafür sorgt, dass der Client beginnend bei der ermittelten Segmentnummer jede Sekunde das nächste Segment von der API anfordert. Ein Zähler ist dafür verantwortlich, die Segmentnummer jede Sekunde um eins hochzuzählen, um die Segmente in korrekter Reihenfolge anzufordern. Nach dem Herunterladen jedes Segments wird die Playlist-Datei basierend auf den vorhandenen Segmenten neu generiert. Die heruntergeladene Playlist kann in diesem Fall nicht übernommen werden, da diese anfangs Segmente enthält, die der Client eventuell nicht heruntergeladen hat, da dieser beim aktuellsten Segment beginnt, das sich am Ende der Playlist befindet. Daher wird nach jedem heruntergeladenen Segment die Playlist-Länge geprüft und entsprechend um das neue Segment erweitert.

Beim Start des Clients wird zunächst ein *HTTPClient*-Objekt erstellt, das benötigt wird, um die API aufzurufen. Folgende Abbildung zeigt den C#-Code:

```
// Create HttpClient for Get-Requests
var handler = new HttpClientHandler()
{
    ServerCertificateCustomValidationCallback =
    HttpClientHandler.DangerousAcceptAnyServerCertificateValidator
};
var client = new HttpClient(handler);
```

Abbildung 73: Erstellung HttpClient, eigener Screenshot

Durch das Anlegen eines Handlers und das Setzen der Option *DangerousAcceptAnyServerCertificateValidator* wird angegeben, dass das Zertifikat für die Verbindung zur API nicht überprüft wird. Da die API beziehungsweise deren Herkunft bekannt ist, gibt es hier keine Sicherheitsrisiken, sodass keine Überprüfung notwendig ist.

10.2.1. Bestimmung der aktuellen Segmentnummer

Zunächst wird betrachtet, wie der Client die Sequenznummer des zuletzt generierten Videosegments ermittelt. Dazu wird die Playlist-Datei im ersten Schritt von der Rest-API abgerufen und innerhalb dessen die Nummer des zuletzt generierten Segments bestimmt. Um dies zu realisieren, wurde die asynchrone Methode *DownloadPlaylist()* geschrieben. Die folgende Abbildung zeigt einen Ausschnitt:

```
var response = await client.GetAsync(restUri);
if (response.IsSuccessStatusCode)
{
    try
    {
        if (File.Exists(apiPlaylist))
        {
            File.Delete(apiPlaylist);
        }
        using (var fs = new FileStream(apiPlaylist, FileMode.Create))
        {
            await response.Content.CopyToAsync(fs);
        }
        if (File.Exists(apiPlaylist))
        {
            List<string> linesList = File.ReadAllLines(apiPlaylist).ToList();
            recentNumber = getLargestSegmentID(linesList);
            Console.WriteLine("Success!");
            File.Delete(apiPlaylist);
        }
```

Abbildung 74: Ausschnitt aus DownloadPlaylist(), eigener Screenshot

Im ersten Schritt wird die Playlist-Datei von der API abgerufen. Dies geschieht mit dem zuvor definierten *HttpClient*-Objekt (*client*) und der Methode *GetAsync()*. Die übergebene Variable *restUri* entspricht dabei einem Uri-Objekt mit dem API-Endpunkt. Die HTTP-Antwort wird schließlich im Objekt *response* gespeichert.

Im Anschluss wird geprüft, ob die Antwort erfolgreich war (*IsSuccessStatusCode*). Ist dies der Fall, wird zunächst innerhalb eines *try*-Blocks geprüft, ob die Playlist bereits existiert. *apiPlaylist* ist dabei eine Zeichenkette, die den Dateipfad zur abgerufenen Playlist auf dem Webserver enthält. Existiert die Datei, wird sie im nächsten Schritt gelöscht, damit sie danach mit der neu abgerufenen Datei ersetzt werden kann. Dazu wird innerhalb eines using-Blocks ein *FileStream*-Objekt angelegt und der Inhalt der HTTP-Antwort, der von der API zurückgegeben wurde, mit *CopyToAsync()* in die Datei kopiert.

Im nächsten Schritt werden alle Zeilen der heruntergeladenen Datei mit *ReadAllLines()* in eine Liste (*linesList*) gelesen. Danach wird die Methode *getLargestSegmentID()* aufgerufen, die die Liste entgegennimmt.

Innerhalb der Methode *getLargestSegmentID()* wird die Liste Element für Element untersucht und jeweils in einer *if*-Abfrage geprüft, ob es sich bei der jeweiligen Zeile um einen Segmentnamen handelt.

Da es sich bei jedem Listeneintrag um eine Zeile der Playlist-Datei, also einer Zeichenkette, handelt und die Segmentnamen immer im Format *stream1_{id}.ts* angelegt sind, kann über die Methoden *StartsWith()* und *EndsWith()* geprüft werden, ob es sich um eine Videodatei handelt. Danach wird die Zeichenkette zerlegt, um die Segmentnummer (*id*) des jeweiligen Segments zu bestimmen.

```
foreach (var line in allLines)
{
    if (line.StartsWith("stream1_") && line.EndsWith(".ts"))
    {
        // determine sequence number of this segment & check if it is correct sequence
        try
        {
            string[] splitLine = line.Split("stream1_");
            string[] numberSegment = splitLine[1].Split(".ts");
            int sequenceNumber = int.Parse(numberSegment[0]);
            if (counter == 0)
            {
                largestSegment = sequenceNumber;
            }
            else
            {
                if (sequenceNumber > largestSegment)
                {
                    largestSegment = sequenceNumber;
                }
```

Abbildung 75: Ermittlung der neusten Segmentnummer, eigener Screenshot

Durch die Methode *Split()* können Zeichenketten an bestimmten Stellen aufgeteilt werden, um Substrings zu erhalten. Um die Segmentnummer zwischen den Zeichenketten „stream1_“ und „.ts“ zu erhalten, wird an dieser Stelle die ursprüngliche Zeile mehrmals geteilt und die Nummer schließlich vom Substring in einen Integer-Wert geparst.

Nachdem die Segmentnummer bestimmt wurde, kann die größte Segmentnummer bestimmt werden. Mittels eines Zählers (*counter*) werden die Segmente durchnummeriert. Handelt es sich um das erste Segment in der Liste, wird die Variable *largestSegment* auf die Segmentnummer dieses Segments gesetzt. Andernfalls wird geprüft, ob die aktuelle Segmentnummer größer ist als die Segmentnummer in der Variable *largestSegment*, und diese entsprechend angepasst.

Nachdem jeder Listeneintrag geprüft wurde, wird in der Variable *largestSegment* die *id* des zuletzt generierten Segments mit der höchsten ID erhalten. Damit weiß der Client, welches Segment zuletzt generiert wurde und im nächsten Schritt angefordert werden kann.

10.2.2. Segmente herunterladen

Nachdem die aktuelle Segmentnummer bestimmt wurde, können die Segmente beginnend bei dieser Nummer von der API angefordert werden. Da die Videosegmente im Dateisystem angelegt werden, bevor sie komplett fertig geschrieben sind, wird jedoch nicht das letzte generierte Segment heruntergeladen, sondern das zweitletzte. Dies stellt sicher, dass die API nicht versucht, eine Datei zu öffnen, die noch von *ffmpeg* geschrieben wird.

Im ersten Schritt wird ein *Timer*-Objekt angelegt und definiert, dass die Methode *DownloadSegmentAtTimeEvent* aufgerufen wird, sobald ein definiertes Intervall, hier 1000 Millisekunden beziehungsweise eine Sekunde, abgelaufen ist.

```
int interval = 1000; // Interval of 1 second
newTimer.Elapsed += new ElapsedEventHandler(DownloadSegmentAtTimeEvent);
newTimer.Interval = interval;
newTimer.Start();
```

Abbildung 76: Timer-Einstellungen, eigener Screenshot

Schließlich wird der Timer mit *Start()* gestartet. Sobald das Intervall abgelaufen ist, wird innerhalb der Methode *DownloadSegmentAtTimeEvent* die Methode *downloadAndUpdate()* aufgerufen, die das nächste Segment herunterlädt und die Playlist erweitert (siehe Anhang C). Schließlich wird der Segmentzähler (*sequenceCounter*) erhöht, damit nach erneutem Ablauf des Intervalls das nächste Segment heruntergeladen werden kann.

Nun soll ein Blick auf die Methode *downloadAndUpdate()* geworfen werden:

```
async Task downloadAndUpdate()
{
    Task.WaitAll(DownloadSegment(sequenceCounter));
    Task.WaitAll(UpdatePlaylist(sequenceCounter));
    Boolean isValid = ValidPlaylist();
```

Abbildung 77: downloadAndUpdate(), eigener Scrreenshot

Die Methode ruft wiederum zwei weitere, asynchrone Methoden auf: *DownloadSegment()* und *UpdatePlaylist()*. Diesen wird jeweils die aktuelle Segmentnummer übergeben. Mit *Task.WaitAll()* wird jeweils auf den Rückgabewert vom Typ *Task* gewartet, um sicherzustellen, dass die Playlist nur erweitert wird, wenn das Segment erfolgreich heruntergeladen wurde und die Playlist im Anschluss mit *ValidPlaylist()* nur validiert wird, wenn sie erfolgreich ergänzt wurde. Falls die Playlist nicht valide ist, wird im Anschluss definiert, dass der *Timer* zurückgesetzt und der Client von vorn beginnt (siehe Anhang C).

Der Code zum Herunterladen eines Segmentes, der sich in der Methode *DownloadSegment()* befindet, ähnelt an sich dem Code zum Abrufen der Playliste (siehe 10.2.1.):

```
var response = await client.GetAsync(restUri);
if (response.IsSuccessStatusCode)
{
    using (var fs = new FileStream(localTarget, FileMode.OpenOrCreate, Fi
    {
        await response.Content.CopyToAsync(fs);
```

Abbildung 78: Herunterladen eines Segments, eigener Screenshot

Bei der Variablen *restUri* handelt es sich um den entsprechenden API-Endpunkt, der die Segmentnummer enthält, sodass der API mitgeteilt wird, welches Segment heruntergeladen werden soll. Danach kann mit *client.GetAsync()* das Segment heruntergeladen werden. Falls dies erfolgreich war, wird ein neuer *FileStream* geöffnet und die HTTP-Antwort in diese lokale Datei (*localTarget*) kopiert.

10.2.3. Playlist aktualisieren

Nun wird betrachtet, wie die Playlist-Datei nach jedem neu heruntergeladenen Segment erweitert wird. Die Playlist wird von *ffmpeg* auf dem Testrechner automatisch bei der Segmentierung des RTMP-Streams generiert. Allerdings ist es nicht möglich, diese regelmäßig von der API abzurufen, ohne das Risiko einzugehen, dass diese entweder Segmente enthält, die auf Client-Seite noch nicht heruntergeladen wurden, oder ältere Segmente enthält, die zu Beginn übersprungen werden, da der Client mit dem aktuellsten Segment beginnt, das sich am Ende der Datei befindet. Aus diesem Grund wird die Playlist-Datei durch den C#-Client angelegt und erweitert. Dazu wurde die Methode *UpdatePlaylist()* geschrieben, die die aktuelle Segmentnummer erwartet:

```
async Task UpdatePlaylist(int segmentID)
{
    currentSegments.Add(segmentID);

    // now delete extra segments from original
    if (currentSegments.Count > playlistLength)
    {
        // remove first segment
        currentSegments.RemoveAt(0);
```

Abbildung 79: UpdatePlaylist()-Teil 1, eigener Screenshot

Im ersten Schritt wird die übergebene Segmentnummer (*segmentID*) einer Liste von Integer-Werten (*currentSegments*) hinzugefügt, die alle Segmentnummern der Playlist-Datei enthält. Diese Liste wird vor Beginn des Programms als globale Variable definiert und dient dazu, im nächsten Schritt die Playlistlänge zu ermitteln und gegebenenfalls zu kürzen. Ist die Länge der Liste größer als die maximale Playlistlänge (*playlistLength*), die global definiert ist, wird das älteste Segment, das an erster Stelle in der Liste steht, gelöscht. Da die Methode *UpdatePlaylist()* nach jedem neu heruntergeladenen Segment aufgerufen wird, kann die Länge der in der Playlist enthaltenen Segmente reduziert werden.

Nun geht es darum, den eigentlichen Inhalt der Playlist-Datei mithilfe der Segment-Liste (*currentSegments*) zu generieren. Die Playlist-Datei hat einen fest definierten Aufbau, der wie folgt aussieht:

```
#EXTM3U
#EXT-X-VERSION:3
#EXT-X-TARGETDURATION:1
#EXT-X-MEDIA-SEQUENCE:657
#EXTINF:1.000000,
stream1_657.ts
#EXTINF:1.000000,
stream1_658.ts
#EXTINF:1.000000,
stream1_659.ts
#EXTINF:1.000000
```

Abbildung 80: .m3u8-Playlist-Datei, eigener Screenshot

Jede Datei beginnt mit vier allgemeinen Informationen, die angeben, um welche Art von Datei es sich handelt, welche Version genutzt wird, wie lang jedes darin enthaltene Segment ist und mit welcher Sequenznummer die Playlist beginnt. Danach werden für jedes Segment zwei Zeilen ergänzt: „#EXTINF:1.000000," und der Dateiname des Segments. Dadurch ergibt sich der Code für den zweiten Teil der *UpdatePlaylist()*-Methode, der den Inhalt generiert und in die Datei schreibt:

```
using (var file = File.Open(tempPlaylist, FileMode.Create, FileAccess.ReadWrite, FileShare.Read))
{
    file.Seek(0, SeekOrigin.End);
    using (var stream = new StreamWriter(file))
    {
        if(currentSegments.Count > 0)
        {
            stream.WriteLine("#EXTM3U\r\n#EXT-X-VERSION:3\r\n#EXT-X-TARGETDURATION:1\n#EXT-X-MEDIA
            for (int i = 1; i < currentSegments.Count; i++)
            {
                int currentsegmentID = currentSegments[i];
                stream.WriteLine("#EXTINF:1.000000,\r\nstream1_" + currentsegmentID + ".ts");
            }
```

Abbildung 81: UpdatePlaylist()-Teil 2, eigener Screenshot

Zunächst wird eine temporäre Playlist-Datei mit dem Namen „stream1_temp.m3u8" angelegt, der in der Variable *tempPlaylist* definiert ist. Danach wird ein *StreamWriter* für die Datei angelegt, mit dem die Datei geschrieben werden kann. Zunächst wird geprüft, ob Segmente in der Liste *currentSegments* vorhanden sind, die Länge also größer Null ist. Da die Datei bei jedem Methodenaufruf neu erstellt wird und damit erstmals leer ist, werden zunächst die ersten vier allgemeinen Zeilen mit *stream.WriteLine()* geschrieben, die Auskunft über die Playlist geben. Im Anschluss wird durch *currentSegments* iteriert, um für jedes Segment in der Liste die zwei Segment-spezifischen Zeilen hinzuzufügen.

Letztlich wird die temporäre Playlist-Datei mit *File.Move()* zur finalen Playlist-Datei verschoben, die der Browser abrufen kann, um die Segmente anzufordern und als Stream abzuspielen. Da die temporäre Datei kontinuierlich von *UpdatePlaylist()* geöffnet und bearbeitet wird, kann es zu Fehlern führen, wenn

der Browser auf diese zugreift. Der Fehler würde den Nutzer darüber informieren, dass die Datei bereits von einem anderem Prozess, in diesem Fall der Methode *UpdatePlaylist()*, verwendet wird. Daher wird die temporäre Datei nach jeder Änderung mit *File.Move()* verschoben:

```
// Move the file
File.Move(tempPlaylist, finalPlaylist, true);
```

Abbildung 82: File.Move(), eigener Screenshot

tempPlaylist ist dabei der Dateipfad zur temporären Datei, *finalPlaylist* der Dateipfad zur finalen Playlist-Datei, die vom Browser abgerufen wird, und *true* gibt an, dass die Datei, falls sie bereits existiert, überschrieben werden soll. Dabei handelt es sich um eine atomare Dateioperation. Das bedeutet, dass die finale Datei entweder den vorherigen Stand der Playlist oder den neuen Stand (nach der Operation) enthält. Der Browser ruft dadurch niemals eine Datei während des Schreibvorgangs ab.

Durch den beschriebenen Ablauf und die genannten Methoden lädt der Client jede Sekunde ein Videosegment herunter, aktualisiert die Playlistdatei und erhöht den Segmentzähler um eins. Der Browser kann parallel dazu die Playlist-Datei abrufen und die verfügbaren Segmente als Stream abspielen.

Neben den bereits beschriebenen API-Abrufen ist der Client ebenfalls dafür verantwortlich, die Videosegmente auf dem Server in Ordnern zu sortieren. Wird das Streaming mit *ffmpeg* beendet oder neu gestartet, beginnt *ffmpeg* erneut damit, die Segmente beginnend bei der Sequenznummer 0 zu nummerieren. Liegen auf dem Server bereits die Segmente mit diesen Sequenznummern ab, werden die Segmente überschrieben. Allerdings sollen Videosegmente auf dem Server gespeichert werden, sodass zum Beispiel der Videostream des vergangenen Tages auch in der Zukunft noch betrachtet werden kann. Daher musste eine Lösung geschaffen werden, sodass die Segmente nicht überschrieben werden. Um dies zu verhindern, erstellt der Client beim Start einen neuen Ordner mit dem aktuellen Datum und der aktuellen Zeit und verschiebt alle existierenden Videosegmente in diesen Ordner. Wenn nun neue Segmente abgerufen werden, sind die alten Segmente im erstellten Ordner gespeichert und werden nicht überschrieben, auch wenn sie denselben Namen besitzen (siehe Anhang C).

10.2.4. Playlist validieren

Falls die Playlist-Datei Segmente enthält, die nicht existieren, oder zum Beispiel ein Segment in der Playlist fehlt, kann es beim Browser-Aufruf zu einem Fehler kommen. Dabei kann es sein, dass die Datei nicht gefunden wird, da sie nicht im Dateisystem des Webservers existiert oder das nächste erwartete Segment nicht in der Playlist definiert ist. Dies könnte auftreten, wenn *ffmpeg* ein Segment überspringt beziehungsweise es einen Fehler beim Anfordern der Datei von der API gibt. Um die Playlist zu validieren, wurde die Methode *ValidPlaylist()* geschrieben:

```
Boolean ValidPlaylist()
{
    Boolean isValid = true;
    int previousNumber = 0;
    for (int counter = 0; counter < currentSegments.Count; counter++)
    {
        // check if file with segmentNr = number exists
        int number = currentSegments[counter];
        string currentFileName = "stream1_" + number + ".ts";
        string path = localPath + currentFileName;
        // if the file does not exist, then return isValid = false
        if (!File.Exists(path))
        {
            isValid = false;
            return isValid;
        }
```

Abbildung 83: ValidPlaylist()-Teil 1, eigener Screenshot

Im ersten Schritt geht es darum, zu prüfen, ob jedes in der Playlist angelegte Segment tatsächlich im Dateisystem existiert. Dazu wird zunächst eine Boolean-Variable *isValid* angelegt, die im ersten Schritt auf *true* gesetzt wird. Außerdem wird eine Integer-Variable *previousNumber* angelegt und auf 0 gesetzt, die im zweiten Teil der Methode Verwendung findet. Danach wird durch die Liste an Segmenten (*currentSegments*) iteriert und für jedes Segment der Dateipfad mit Dateinamen als *string* zusammengesetzt, um im Anschluss mit *File.Exists()* zu prüfen, ob die Datei mit der jeweiligen Segmentnummer existiert. Ist dies nicht der Fall, wird *isValid* auf *false* gesetzt und diese Variable zurückgegeben.

Falls das Segment existiert, wird im nächsten Schritt geprüft, ob die in der Playlist enthaltenen Segmente in aufsteigender Reihenfolge sind beziehungsweise ein Segment übersprungen wurde:

```
if (counter == 0)
{
    previousNumber = number;
}
else
{
    if (number == (previousNumber + 1))
    {
        previousNumber = number;
    }
    else
    {
        isValid = false;
        return isValid;
    }
}
```

Abbildung 84: ValidPlaylist()-Teil 2, eigener Screenshot

Falls es sich um die erste Datei in der Playlist-Datei handelt, die Variable *counter* also Null ist, wird die Variable *previousNumber* auf die aktuelle Segmentnummer gesetzt. Ansonsten wird geprüft, ob es sich bei der aktuellen Segmentnummer um die vorherige Nummer (*previousNumber*) um eins erhöht handelt, was von einer validen Playlist-Datei erwartet wird. Falls dies der Fall ist, wird *previousNumber* auf die aktuelle Segmentnummer gesetzt. Ist dies nicht der Fall, kann angenommen werden, dass ein Segment in der Playlist-Datei fehlt. In diesem Fall wird *isValid* auf *false* gesetzt und zurückgegeben.

Läuft die Methode bis zum Ende durch, ohne einen Fehler zu finden, bleibt die Variable *isValid* auf *true* und wird zurückgegeben.

An der Stelle, an der die Methode *ValidPlaylist()* aufgerufen wird, wird definiert, was passiert, wenn die Playliste valide beziehungsweise invalide ist. In diesem Fall soll das Programm normal weiterlaufen, wenn die Playlist valide ist, die Methode also *true* zurückgibt, und neugestartet werden, wenn die Playlist invalide ist, also *false* zurückgegeben wird. Dabei werden alle Segmente in der Liste *currentSegments* gelöscht und der *Timer* wird neu gestartet (siehe Anhang C). Dann beginnt der Client wieder mit dem Download der Playliste und der Bestimmung des neusten Segments (siehe Anhang C).

10.3. Docker-Images erstellen

Um die neue Architektur auf dem Testsystem umzusetzen, mussten zwei Docker-Images erstellt und die Container auf die entsprechenden Systeme übertragen werden. Ein Container soll dabei einen *nginx*-RTMP-Server in der Linux-VM laufen lassen. Dieser befindet sich auf dem Testrechner, empfängt den von *ffmpeg* generierten RTMP-Stream und erlaubt dem zweiten *ffmpeg*-Skript den RTMP-Stream abzurufen und zu segmentieren. Der zweite Container, der erstellt wird, ist dafür zuständig, den C#-Rest-Client in der Cloud zu hosten, der in regelmäßigen Abständen die Rest-API auf dem Windows-Testrechner aufruft und die Segmente herunterlädt und speichert.

Der Docker-Container für den RTMP-Server wurde bereits installiert und in Abschnitt 9.1. ausführlich beschrieben, sodass dieser nicht nochmals erläutert wird. Im Folgenden soll der für die neue Architektur relevante Container für den C#-Rest-Client näher betrachtet werden.

Folgende Abbildung zeigt das Dockerfile, auf dem der Container basiert:

```
FROM nginx:latest

RUN apt-get update
RUN apt install -y python3-venv python3-pip
RUN pip install supervisor

RUN apt install -y wget
RUN wget https://packages.microsoft.com/config/debian/11/packages-microsoft-prod.deb -O packages-microsoft-prod.deb
RUN dpkg -i packages-microsoft-prod.deb
RUN rm packages-microsoft-prod.deb
RUN apt-get update
RUN apt-get install -y dotnet-sdk-6.0

COPY src/html /usr/share/nginx/html
ADD rest-client /usr/share/restclient

ADD supervisord.conf /usr/local/etc/supervisord.conf
CMD ["supervisord"]
```

Abbildung 85: Dockerfile für den C#-Rest-Client, eigener Screenshot

Das Dockerfile basiert auf dem *nginx*-Image, das auf Dockerhub zu finden ist [129]. Im nächsten Schritt wird ein Update durchgeführt und der Paketmanager *pip* installiert. Dieser wird daraufhin benötigt, um *supervisor* zu installieren.

Im Anschluss muss .NET 6.0 installiert werden, um den C#-Rest-Client innerhalb des Containers ausführen zu können. Dazu wird mit *apt install -y wget* das Programm *wget* installiert, um im Anschluss den Microsoft-Signierungsschlüssel zu den vertrauenswürdigen Schlüsseln hinzuzufügen und das Paket-Repository herunterzuladen [130]. Danach kann die .NET SDK, in diesem Fall die Version 6.0, mit *apt-get install -y dotnet-sdk-6.0* heruntergeladen werden [130].

Daraufhin werden die notwendigen HTML-, CSS- und JavaScript-Dateien im Ordner *src/html* in den Container kopiert. Als nächstes werden das .NET-Projekt mit dem C#-Rest-Client sowie die *supervisord.conf*-Datei an die entsprechende Stelle im Container kopiert. Letztlich wird *supervisor* gestartet, das beim Start des Containers dafür zuständig ist, den *nginx*-Webserver und den C#-Rest-Client zu starten. Folgende Abbildung zeigt die *supervisord.conf*-Datei:

```
[supervisord]
nodaemon=true

[program:nginx]
command=nginx
startsecs=1
startretries=3
stdout_logfile=/usr/share/nginx/html/output_nginx.log
stderr_logfile=/usr/share/nginx/html/err_nginx.log

[program:dotnet]
directory=/usr/share/restclient/TestApp_RestClient
command=dotnet run
startsecs=1
startretries=3
stdout_logfile=/usr/share/restclient/output_dotnet.log
stderr_logfile=/usr/share/restclient/err_dotnet.log
```

Abbildung 86: supervisord.conf-Datei, eigener Screenshot

Das erste Programm startet mit dem Kommando *nginx* den *nginx*-Webserver. Das zweite Programm startet den C#-Rest-Client. Dazu wird zunächst mit *directory* das Verzeichnis angegeben, in den das .NET-Projekt bei der Erstellung des Docker-Images kopiert wurde. Mit dem Kommando *dotnet run* wird das Projekt schließlich gestartet und es beginnt, die Videosegmente von der API auf dem Testrechner anzufordern und auf dem Webserver abzulegen.

10.4. Übergabe des API-Endpunkts

Da die C#-Rest-API in der Zukunft auf allen TBM-Rechnern laufen soll, muss es für jeden TBM-Rechner auch einen Client geben, der die entsprechende API aufruft. Daher wurde der Client so angepasst, dass der API-Endpunkt nicht direkt im Code angegeben wird, sondern über eine Umgebungsvariable beim Starten des Containers übergeben wird. Dadurch kann derselbe Container mehrmals in der Cloud laufen, wobei jeder für einen anderen TBM-Rechner zuständig ist.

Um den API-Endpunkt für einen bestimmten TBM-Rechner zu übergeben, kann der Docker-Container wie folgt gestartet werden:

```
docker run -p 4000:80 -e "API=http://IP-Adresse:portNr" image-ID
```

Abbildung 87: Container-Start mit Umgebungsvariable, eigener Screenshot

Der Container wird mit *docker run* gestartet. Dabei wird mit *-e* eine Umgebungsvariable definiert und in den Container übergeben. In diesem Fall handelt es sich um die Variable mit dem Namen *API*, die die IP-Adresse und die Portnummer der entwickelten C#-Rest-API enthält.

Mit *-p* können Portnummern gemappt werden. In diesem Beispiel wird die interne Portnummer 80 auf die Portnummer 4000 gemappt, sodass der *nginx*-Server außerhalb des Containers unter der Portnummer 4000 aufgerufen werden kann. Letztlich muss noch die Image-ID angegeben werden.

Um die Umgebungsvariable *API* im C#-Code auszulesen und als API-Endpunkt zu nutzen, wurde der C#-Rest-Client um folgende Zeilen ergänzt:

```
var envValue = System.Environment.GetEnvironmentVariable("API");
string address = envValue.ToString() + "/api/segment/";
```

Abbildung 88: Umgebungsvariable auslesen, eigener Screenshot

Mit *System.Environment.GetEnvironmentVariable("API")* wird die Umgebungsvariable vom System ausgelesen. Danach kann die Adresse aus den verschiedenen Zeichenketten und der ausgelesenen Variablen zusammengesetzt werden, um den API-Endpunkt im Format *http://ip-address:portNr/api/segment/* zu erhalten. Dadurch kann derselbe C#-Code für jeden beliebigen TBM-Rechner beziehungsweise für jede darauf laufende API eingesetzt werden, ohne den Code für jeden Rechner manuell anpassen zu müssen. Dies ist wichtig, wenn es darum geht, den Prototypen mit wenig Aufwand für möglichst alle laufenden Projekte einsetzen zu können.

Zusammenfassend lässt sich sagen, dass die Verbindung zwischen Testrechner und Webserver nach der Architektur-Erweiterung erfolgreich hergestellt werden konnte. Dadurch können die auf dem Testrechner abgelegten HLS-Segmente per API abgerufen und auf dem Webserver gespeichert werden. Die vom Rest-Client generierte Playlist lässt sich schließlich vom Browser (Client) abrufen, um die einzelnen Segmente herunterzuladen und als Videostream wiederzugeben.

11. Automatisierung

Nachdem die neue Architektur umgesetzt wurde, ging es darum, den Prototyp in das Test-*CONNECTED* auf Kubernetes zu integrieren. Außerdem wird ein Blick darauf geworfen, wie die Software automatisiert auf den Tunnelbohrmaschinen installiert und gestartet werden kann. Dies ist notwendig, um zu verhindern, dass die benötigte Software für jede Maschine einzeln übertragen und gestartet werden muss. Außerdem soll es möglich sein, die Software im Hintergrund auf den TBM-Rechnern laufen zu lassen.

11.1. Kubernetes-Integration

Im ersten Schritt ging es darum, den erstellten Docker-Container mit dem C#-Rest-Client in Kubernetes zu integrieren. In diesem Abschnitt wird es darum gehen, zunächst Kubernetes vorzustellen und im Anschluss zu beschreiben, wie der Docker-Container, der den im Rahmen dieser Thesis entwickelten C#-Rest-Client enthält, in Kubernetes integriert wurde, um dort in einem Testsystem automatisiert zum Laufen gebracht zu werden.

11.1.1. Kubernetes-Grundlagen

Kubernetes ist ein open-source-Tool, das es ermöglicht, Container, wie zum Beispiel Docker-Container, innerhalb einer Cluster-Umgebung laufen zu lassen [131]. Das Tool wird auch als „containerzentrierte Managementumgebung“ [132] bezeichnet und erleichtert „die Bereitstellung, Skalierung und Verwaltung von Anwendungen“ [132]. Kubernetes unterstützt dabei verschiedene Containerarten [133], allerdings sind für diese Arbeit lediglich Docker-Container relevant.

Kubernetes basiert auf sogenannten ‚Clustern‘. Ein Cluster ist eine Sammlung hochverfügbarer, vernetzter Computer, die als einzelne Einheit fungieren [134]. Kubernetes koordiniert diese und ermöglicht es, containerisierte Applikationen innerhalb eines Clusters auszuführen, ohne diese an eine physische Maschine zu binden [134]. Die Distribution sowie die zeitliche Planung können von Kubernetes effizient automatisiert werden. Folgende Abbildung zeigt die Architektur eines Kubernetes-Clusters und die zugehörigen Komponenten, auf die im Folgenden eingegangen wird:

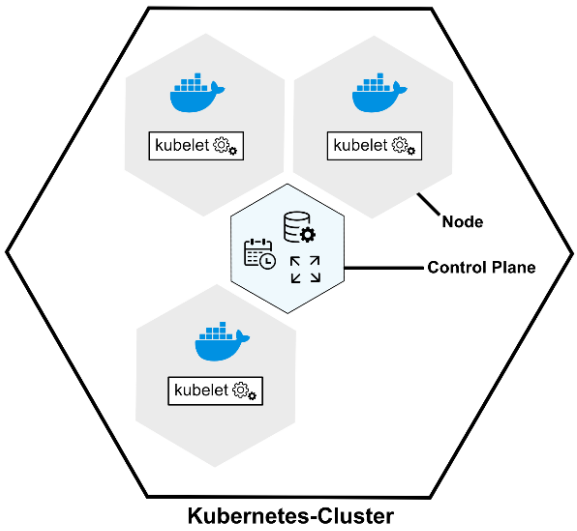

Abbildung 89: Kubernetes-Cluster, eigene Darstellung in Anlehnung an [134]

Innerhalb eines Clusters gibt es zwei verschiedene Arten von Ressourcen: die Control Plane und eine Sammlung an sogenannten ‚Nodes' [134]. Die Control Plane ist dafür verantwortlich, das Cluster zu verwalten [134]. Dazu gehören die Koordination aller Aktivitäten innerhalb des Clusters, wie zum Beispiel das Aufstellen eines Zeitplans für Applikationen, und die Sicherstellung des Soll-Zustands der im Cluster laufenden Applikationen [134]. Weiterhin kümmert sich die Control Plane um die Skalierung und das Ausrollen neuer Updates [134]. Ein Node hingegen ist eine virtuelle oder physische Maschine, der sogenannte ‚Pods' zugewiesen werden [133].

Ein Pod ist eine Sammlung an Containern, die eine Einheit bilden [133]. Dabei können mehrere Applikationen innerhalb eines Pods laufen oder eine Applikation innerhalb eines Pods auf mehrere Container verteilt sein. In der Regel werden jedoch verschiedene Container für unterschiedliche Applikationen eingesetzt [133]. Alle Container und die darin enthaltenen Applikationen teilen sich ein gemeinsames Dateisystem und eine IP-Adresse, jedoch verfügt jede Applikation über eine eigene Portnummer [133]. Der Einsatz von Pods vereinfacht das Teilen von Ressourcen und die Kommunikation. So können stark verknüpfte Applikationen innerhalb mehrerer Container auf demselben Pod laufen [133].

Neben Pods verfügt jeder Node auch über ein sogenanntes ‚*Kubelet*'. Dabei handelt es sich um einen Service, der den Node verwaltet [131]. Dieser ist auch für die Kommunikation zwischen dem Node und der Control Plane verantwortlich [134].

Generell werden Nodes in *Master Nodes* und *Worker Nodes* unterschieden [133]. Worker Nodes dienen dazu, Container auszuführen [135]. Master Nodes hingegen sind dafür verantwortlich, den Worker Nodes Anweisungen zu geben. So sind sie zum Beispiel dafür verantwortlich, die Container auf die richtigen Worker Nodes zu verteilen. Ein Cluster kann über einen oder mehrere Master Nodes verfügen [135]. Theoretisch ist es möglich, dass ein Master Node mehrere Tausend Worker Nodes verwaltet [135].

Beim Start von Applikationen in Kubernetes wird der Control Plane mitgeteilt, welche Container gestartet werden sollen [134]. Die Control Plane sorgt im nächsten Schritt dafür, dass die entsprechenden Container auf den Nodes zum Laufen gebracht werden. Kubernetes stellt die Kubernetes-API zur Verfügung, die von der Control Plane offengelegt wird. Über diese läuft nicht nur die Kommunikation zwischen der Control Plane und den Nodes, sondern auch die Kommunikation zwischen Endnutzern und dem Cluster [134].

Zwei weitere wichtige Kubernetes-Komponenten sind *Services* und *Ingress*. Ein Service ist eine externe Schnittstelle, die einem oder mehreren Pods zugewiesen werden kann und Endpunkte bietet, über die die Applikationen innerhalb der Pods aufgerufen werden können [133]. Dadurch können externe Clients den Service mit dem jeweiligen Namen und der Portnummer der Applikation aufrufen, ohne über die vom Service verwalteten Pods Bescheid wissen zu müssen. Der Service ist dann dafür verantwortlich, die Anfragen an den jeweiligen Pod weiterzuleiten [133].

Ein *Ingress* (engl. Eingang) ist hingegen eine Art Reverse-Proxy und dafür verantwortlich, die URL von Proxyserver-Anfragen internen URLs zuzuweisen [135]. Dadurch werden externe Anfragen an den entsprechenden Service weitergeleitet [131].

Zu den Vorteilen von Kubernetes gehört unter anderem die Tatsache, dass Applikationen in kleinere, besser verwaltbare und skalierbare Komponenten aufgeteilt werden können [133]. Außerdem sind Cluster fehlertolerant, sodass zum Beispiel beim Ausfall eines Pods eine Kopie des Pods automatisch gestartet wird. Weiterhin kann definiert werden, wie viele Kopien eines Pods parallel laufen sollen, was für eine horizontale Skalierung und Fehlertoleranz sorgt [133]. Letztlich dient Kubernetes dazu, Ressourcen besser und effizienter zu nutzen und Komponenten aufzuteilen, was der Arbeitsteilung dient [133].

Aus diesen Gründen geht es im nächsten Schritt darum, den erstellten Docker-Container innerhalb von Kubernetes zum Laufen zu bringen. Da das Tool bereits von der Herrenknecht AG eingesetzt wird, wird die Installation der Kubernetes-Umgebung nicht thematisiert.

11.1.2. Integration des Docker-Containers

In diesem Abschnitt werden die notwendigen Schritte zur Integration des Docker-Containers in die Kubernetes-Testumgebung der Herrenknecht-AG behandelt, damit der C#-Rest-Client automatisiert zum Laufen gebracht werden kann.

Für die Integration des Docker-Containers in Kubernetes werden einige *Azure-DevOps*-Produkte genutzt. *Azure DevOps* ist eine Sammlung an Diensten, die bei der Softwareentwicklung unterstützen [136]. Sie dienen dazu, „Produkte schneller zu erstellen und zu verbessern“ [136], und werden bei der Herrenknecht AG eingesetzt. Die Dienste, die für dieses Projekt zum Einsatz kommen, sind einerseits *Azure Repos* und andererseits *Azure Pipelines*. *Azure Repos* dient der Quellcodeverwaltung und *Azure*

Pipelines unterstützt bei der „kontinuierliche[n] Integration und Bereitstellung“ [136] von Applikationen. Das Tool ermöglicht es, Codeprojekte automatisch zu erstellen und zu testen [137]. Dabei werden folgende Methoden verwendet: *Continuous Integration*, *Continuous Delivery* und *Continuous Testing*. *Continuous Integration* (CI) ist ein Prozess, bei dem Code automatisiert, zusammengeführt und getestet werden kann [137]. Dies dient einerseits der frühzeitigen Fehlererkennung und andererseits der Qualitätssicherung. Beim *Continuous Delivery* (CD) wird der Code in verschiedenen „Test- und Produktionsumgebungen bereitgestellt“ [137]. Diese Systeme erzeugen sogenannte ‚Artefakte‘, die genutzt werden, um „neue Versionen und Fixes für vorhandene Systeme zu veröffentlichen“ [137]. Dies dient der Erhöhung der Qualität. Letztlich ist das *Continuous Testing* dafür zuständig, Testworkflows zu automatisieren. Kontinuierliche Tests sind wichtig, um sicherzustellen, dass die Applikationen nach jeder neu veröffentlichten Version weiterhin funktionieren [137].

Um den Docker-Container innerhalb von Kubernetes zum Laufen zu bringen, mussten zunächst das erstellte Dockerfile (siehe 10.3.) und die dazugehörigen Dateien, darunter der Code des C#-Rest-Clients, in ein Repository, in diesem Fall ein *Azure-DevOps*-Repository, hochgeladen werden.

Im nächsten Schritt konnte die Kubernetes-Konfiguration durchgeführt werden, um zu definieren, wie der Docker-Container in der Kubernetes-Umgebung laufen soll. Dazu wird eine Konfigurationsdatei mit der Endung *.yml* benötigt. Diese verfügt über drei verschiedene Abschnitte: *Deployment*, *Service* und *Ingress*. Im Folgenden wird ein Blick auf die verschiedenen Einstellungen geworfen, die hier angegeben werden.

Der erste Abschnitt ist *Deployment*. In diesem werden Angaben zum Start des Containers gemacht.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: streaming-test
  namespace: herrenknecht-connected
spec:
  replicas: 1
  selector:
    matchLabels:
      app: streaming-test
```

Abbildung 90: Config-Deployment 1, eigener Screenshot

Dem Abschnitt werden zunächst ein Name (*streaming-test*) sowie ein Namensraum (*herrenknecht-conntected*) vergeben. Außerdem wird angegeben, wie viele Kopien (*replicas*) des Containers laufen sollen, in diesem Fall zum Testen nur eine. Die Applikation innerhalb des Containers erhält den Namen *streaming-test* (siehe Abbildung 90).

```
spec:
 nodeSelector:
  "kubernetes.io/os": linux
 containers:
  - name: streaming-test
   image: image-path
   imagePullPolicy: IfNotPresent
   resources:
    requests:
     memory: "256Mi"
```

Abbildung 91: Config-Deployment 2, eigener Screenshot

Im nächsten Schritt wird angegeben, welches Betriebssystem verwendet werden soll. In diesem Fall wird Linux verwendet. Danach werden Angaben zum Container gemacht, der in Kubernetes integriert werden soll. Neben dem Namen wird der Pfad zum Repository angegeben, in dem sich das Dockerfile und die benötigten Code-Dateien befinden. Weiterhin kann angegeben werden, wie viel Speicherplatz die Applikation maximal in Anspruch nehmen darf.

```
   ports:
    - containerPort: 80
   env:
    - name: API
     value: "path/ip-address:port"
terminationGracePeriodSeconds: 5
```

Abbildung 92: Config-Deployment 3, eigener Screenshot

Danach kann definiert werden, auf welchem Port die Applikation innerhalb des Containers erreichbar sein soll. Hier ist dies der Port 80. Letztlich wird eine Umgebungsvariable mit dem Namen „API“ gesetzt, die dem Container beim Start übergeben wird. Beim manuellen Start des Docker-Containers wird sie im Kommando mitgegeben (siehe 10.4.). Da der Container hier automatisch gebaut wird, muss die Variable API mit der IP-Adresse des Rechners, auf dem die entwickelte C#-Rest-API läuft, zusammen mit der Portnummer in der Konfiguration festgelegt werden. Die Umgebungsvariable wird benötigt, damit der C#-Rest-Client die API auf dem gewünschten Rechner aufrufen kann. Sie wird im C#-Code ausgelesen (siehe 10.4.).

Der nächste Abschnitt definiert einen *Service*:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: streaming-test
  namespace: herrenknecht-connected
spec:
  selector:
    app: streaming-test
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
```

Abbildung 93: Config-Service, eigener Screenshot

In diesem Abschnitt werden, wie im Abschnitt zuvor, der Name und der Namensraum definiert. Zudem wird angegeben, welches Protokoll verwendet wird, in diesem Fall das Transportprotokoll TCP. Letztlich kann angegeben werden, welcher Port innerhalb des Containers genutzt wird und auf welchem Ziel-Port dieser von außerhalb erreichbar sein soll.

Letztlich folgt der Abschnitt *Ingress*. Dieser definiert, wie interne Services von außerhalb des Clusters erreicht werden können [138].

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  annotations:
    kubernetes.io/ingress.class: nginx
    nginx.ingress.kubernetes.io/proxy-buffer-size: 128k
    nginx.ingress.kubernetes.io/rewrite-target: /$2
  name: streaming-test
  namespace: herrenknecht-connected
```

Abbildung 94: Config-Ingress 1, eigener Screenshot

Dazu wird zunächst die Ingress-Klasse mit *nginx* definiert, da der Container einen *nginx*-Webserver enthält. Weiterhin werden die Puffergröße, der Name sowie der Namensraum angegeben (siehe Abbildung 94).

```yaml
spec:
 rules:
  - host: host-url
   http:
    paths:
     - backend:
       service:
        name: streaming-test
        port:
         number: 80
      path: /streaming-test(/|$)(.*)
      pathType: Prefix
 tls:
  - hosts:
    - host-url
   secretName: secret-url
```

Abbildung 95: Config-Ingress 2, eigener Screenshot

Im nächsten Schritt werden die Abschnitte *rules* und *tls* definiert. *rules* definiert, wie Anfragen weitergeleitet werden [138]. In diesem Fall wird eine HTTP-Regel für einen gegebenen Host definiert. Hier wird angegeben, dass alle Anfragen mit dem definierten Host (*host*) und Pfad (*path*) an den Service mit dem Namen *streaming-test* und der Portnummer 80 weitergeleitet werden.

Der Abschnitt *tls* definiert, dass der Ingress verschlüsselt ist [138]. Dazu werden die Hosts (*hosts*), in diesem Fall nur ein Host, sowie der Name des sogenannten ‚*Secrets*' (*secretName*) angegeben. Ein *Secret* enthält einen TLS-Schlüssel und ein TLS-Zertifikat, die für die Verbindung benötigt werden [138].

Die vollständige Konfigurationsdatei (*streaming-test.yml*) ist dem Anhang D zu entnehmen.

Im nächsten Schritt konnte eine Azure-Pipeline angelegt werden, die dafür zuständig ist, den Docker-Container zu bauen. Danach konnte der Docker-Container in Kubernetes ausgerollt werden, was über die Kommandozeile geschehen kann.

Nachdem der Container in Kubernetes ausgerollt wurde, konnte die konfigurierte URL aufgerufen werden, um die erstellte HTML-Seite aufzurufen und den Videostream im Videoplayer anzusehen.

11.2. Installations- & Start-Automatisierung

In diesem Abschnitt soll ein Blick darauf geworfen werden, welche Tools bei der Herrenknecht AG eingesetzt werden, um Software automatisch auf Tunnelbohrmaschinen zu installieren und zu starten.

Da es sich in dieser Thesis lediglich um einen Prototyp handelt, der noch nicht auf einer aktiven Tunnelbohrmaschine eingesetzt wird, wird die Automatisierung der Installation nicht für den Prototypen umgesetzt, sondern lediglich beschrieben, welche Software dafür verwendet werden kann und wie dies in der Zukunft umgesetzt werden kann.

11.2.1. Tool 1: *Ansible*

Um Software automatisch auf den Rechnern der Tunnelbohrmaschinen zu installieren, wird das Tool *Ansible* genutzt. *Ansible* ist ein open-source-IT-Automatisierungssystem [139]. Es ist unter anderem dafür zuständig, Konfigurationen zu verwalten und Applikationen zu starten [139]. Dabei kann die Verwaltung entfernter Systeme mithilfe des Tools automatisiert sowie der Zustand dieser kontrolliert werden [140].

Folgende Abbildung zeigt die *Ansible*-Architektur:

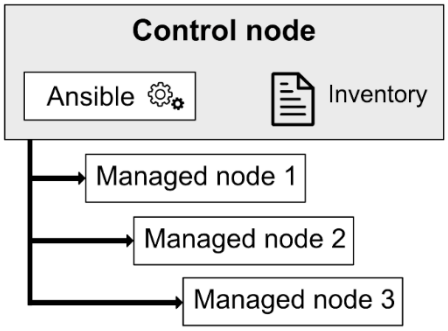

Abbildung 96: Ansible-Architektur, eigene Darstellung in Anlehnung an [140]

Ansible unterscheidet drei verschiedene Komponenten: *control node*, *managed node* und *inventory* [140]. Die *control node* ist auch der Host-Rechner, auf dem die eigentliche *Ansible*-Software läuft [140]. Dabei muss es sich um einen UNIX-Rechner handeln, auf dem Python installiert ist [141]. Dieser ist dafür verantwortlich, sogenannte ‚*managed nodes*' zu verwalten. Dabei handelt es sich um eine Vielzahl entfernter Maschinen oder Geräte [141]. Die verwalteten Nodes (*managed nodes*) benötigen im Gegensatz zur *control node* keine *Ansible*-Installation, allerdings muss auf ihnen ebenso Python installiert sein, um den *Ansible library code* ausführen zu können [141]. Die dritte Komponente ist das sogenannte ‚*inventory*'. Dies ist eine Liste von *managed nodes*, die in einer logischen Struktur organisiert sind.

Um die Software nutzen zu können, müssen zunächst auf einem Host-Rechner die *Ansible*-Software und Python (mindestens die Version 3.9) installiert werden [141]. Im nächsten Schritt kann ein *inventory* angelegt werden, indem die IP-Adressen beziehungsweise die *fully qualified domain names* (FQDN) der entfernten Rechner angegeben werden [140]. Danach müssen *ssh*-Verbindungen (siehe 9.1.) zu den entfernten Rechnern aufgebaut werden, indem der *ssh*-Schlüssel des Hosts auf den entfernten Rechnern gespeichert wird [140].

Nachdem die Verbindungen aufgebaut wurden, können verschiedene *Ansible*-Module eingesetzt werden, um Vorgänge und Aufgaben auf den entfernten Rechnern zu automatisieren [142]. Dazu wird eine sogenannte ‚Playbook-Datei' mit der Dateiendung *.yml* angelegt, die, ähnlich wie ein Dockerfile, Anweisungen enthält, die auf dem entfernten System durchgeführt werden [143]. Folgende Abbildung zeigt eine beispielhafte Playbook-Datei:

```
- name: Install a list of packages
  yum:
    name:
      - nginx
      - postgresql
      - postgresql-server
    state: present
```

Abbildung 97: Beispiel-Playbook-Datei (.yml), Quelle: [142]

Diese Playbook-Datei definiert zum Beispiel, dass die unter *yum* definierten Software-Pakete *nginx*, *postgresql* und *postgresql-server* auf dem entsprechenden System installiert werden sollen [142].

Um die notwendigen Komponenten des im Rahmen dieser Thesis entwickelten Prototypen auf den TBM-Rechnern zu installieren, könnte auf diese Weise eine Playbook-Datei mit den Installationsschritten geschrieben werden.

Dabei ist es möglich, mit dem Schlüsselwort *copy* zu definieren, dass Dateien von einer bestimmten Quelle, zum Beispiel dem Host-Rechner, auf die entfernten Rechner kopiert werden [142]. Das Schlüsselwort *command* ermöglicht es Kommandos auszuführen. Neben diesen genannten Schlüsselworten beziehungsweise Modulen gibt es eine Vielzahl weiterer Module, mit deren Hilfe die Softwareinstallation der Prototyp-Komponenten auf den TBM-Rechnern automatisiert werden kann.

11.2.2. Tool 2: *NSSM*

Nachdem die Automatisierung der Softwareinstallation betrachtet wurde, soll nun ein Blick auf den automatischen Start der installierten Komponenten auf den TBM-Rechnern geworfen werden. Um Programme automatisiert auf den Rechnern der Tunnelbohrmaschinen starten zu können, wird bei der Herrenknecht AG das Tool *NSSM* eingesetzt.

NSSM steht für ‚*Non-Sucking Service Manager*' und dient der Verwaltung von Windows-Services [144]. Es unterstützt dabei, bestimmte Aufgaben zu automatisieren und sicherzustellen, dass Services kontinuierlich laufen [145]. Dabei kann NSSM eingesetzt werden, um Services als Hintergrundprozesse, zum Beispiel beim Rechnerstart, zu starten, sodass keine Benutzerinteraktion zum Starten der Services benötigt wird [144]. Außerdem können Skripte als Service laufen gelassen werden. Dazu muss lediglich der Pfad zur entsprechenden .bat- oder cmd-Datei angegeben werden und es wird automatisch ausgeführt [145]. *NSSM* kann auch konfiguriert werden, um zum Beispiel beim Absturz von Applikationen zu versuchen, diese neu zu starten [145].

Um NSSM zu nutzen, wird die Software einmalig heruntergeladen und in einem bestimmten Ordner auf dem entsprechenden Windows-Rechner abgelegt [146]. Im nächsten Schritt kann wie bei der *ffmpeg*-Installation (siehe 8.1.1.) der Speicherort der Umgebungsvariable PATH hinzugefügt werden, um von überall aus erreichbar zu sein. Schließlich kann das Tool gestartet werden [146].

Nun kann mittels der Kommandozeile oder über das *NSSM*-Konfigurationsfenster ein Service installiert werden [146].

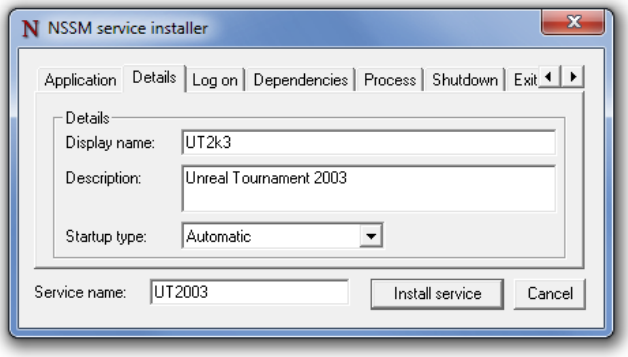

Abbildung 98: NSSM service installer, Quelle: [146]

Um ein Service über das *NSSM*-Konfigurationsfenster zu installieren, gibt es verschiedene Tabs, über die die Konfiguration durchgeführt werden kann.

Unter dem *Application*-Tab wird der Pfad zur Applikation oder zum Skript angegeben, das als Service gestartet werden soll [146]. Unter *Details* werden für den Service ein Name, eine Beschreibung und eine Startup-Art definiert. Weitere Tabs ermöglichen die Konfiguration von Abhängigkeiten, Prozess-Prioritäten, Einstellungen zum Beenden des Services und weiteren Aspekten [146].

Die Konfiguration kann auch über die Kommandozeile durchgeführt werden. Dabei können Services wie über die GUI installiert, verwaltet und bearbeitet werden.

```
nssm start <servicename>

nssm stop <servicename>

nssm restart <servicename>
```

Abbildung 99: Beispiel-Kommandos zum Start, Stop & Restart von NSSM, Quelle: [146]

Für den automatischen Start der Prototyp-Komponenten auf den entfernten TBM-Rechnern kann in der Zukunft *NSSM* eingesetzt werden. Dazu können die *ffmpeg*-Skripte sowie die C#-Rest-API, die als *.exe*-Datei auf die Testrechner übertragen werden kann, jeweils als eigener Service definiert und im Hintergrund gestartet werden. Um zu vermeiden, dass alle drei Komponenten bei jedem Rechnerstart durch den Nutzer gestartet werden müssen, kann der entsprechende Parameter für den automatischen Start gesetzt werden [146]. Außerdem wäre eine sinnvolle Einstellung, dass die Services beim Abbruch nach bestimmter Zeit neu gestartet werden, um das Streaming möglichst kontinuierlich und ohne Unterbrechungen auf *CONNECTED* zur Verfügung stellen zu können.

12. Testen

In diesem Kapitel wird einerseits getestet, welche *ffmpeg*-Optionen genutzt werden können, um eine adaptive Bitrate (ABR) anzubieten. Außerdem werden einige Latenztests durchgeführt, um zu messen, wie hoch die Verzögerungsrate des implementierten Prototyps durchschnittlich ist.

12.1. HTTP-Live-Streaming mit adaptiver Bitrate

Für den Prototyp wurde auf das adaptive Bitrate-Streaming verzichtet, um den Kompressionsaufwand und dadurch die Latenz zwischen der Bildschirmaufnahme und der Wiedergabe des Streams auf den Clients möglichst gering zu halten. Daher wird lediglich eine Version jedes HLS-Segments in der ursprünglichen Auflösung des Eingangssignals generiert und auch nur eine Playlist-Datei erzeugt. Dennoch bietet *ffmpeg* die Möglichkeit, adaptives Bitrate-Streaming zu nutzen. Dabei kann im Kommando definiert werden, dass dasselbe Eingangssignal in verschiedenen Auflösungen beziehungsweise Bitraten zur Verfügung gestellt werden soll, sodass der Client in Abhängigkeit von der verfügbaren Bandbreite die für ihn passende Variante herunterladen kann (siehe 5.4.3.). In diesem Fall wird für jede definierte Variante eine eigene Playlist erstellt und eine Master-Playlist generiert, die Auskunft über die verfügbaren Varianten gibt.

Um dies zu testen, wurde das von Vijayanagar [147] erstellte *ffmpeg*-Kommando angepasst, sodass der eingehende RTMP-Stream in drei verschiedene HLS-Varianten kodiert und bereitgestellt wird:

```
ffmpeg -i rtmp://localhost:1935/live/stream1 ^
    -filter_complex "[0:v]split=3[v1][v2][v3]; [v1]copy[v1out]; ^
    [v2]scale=w=1280:h=720[v2out]; [v3]scale=w=640:h=360[v3out]" ^
    -map [v1out] -c:v:0 libx264 -x264-params "nal-hrd=cbr:force-cfr=1" ^
    -b:v:0 5M -maxrate:v:0 5M -minrate:v:0 5M -bufsize:v:0 10M ^
    -preset fast -g 24 -sc_threshold 0 -keyint_min 24 ^
    -map [v2out] -c:v:1 libx264 -x264-params "nal-hrd=cbr:force-cfr=1" ^
    -b:v:1 3M -maxrate:v:1 3M -minrate:v:1 3M -bufsize:v:1 3M ^
    -preset fast -g 24 -sc_threshold 0 -keyint_min 24 ^
    -map [v3out] -c:v:2 libx264 -x264-params "nal-hrd=cbr:force-cfr=1" ^
    -b:v:2 1M -maxrate:v:2 1M -minrate:v:2 1M -bufsize:v:2 1M ^
    -preset fast -g 24 -sc_threshold 0 -keyint_min 24 ^
    -f hls -hls_time 1 -hls_flags delete_segments -hls_list_size 20 ^
    -hls_segment_filename ./hls/stream_%v/data%02d.ts ^
    -master_pl_name master.m3u8 ^
    -var_stream_map "v:0 v:1 v:2" ./hls/stream_%v/stream_.m3u8
```

Abbildung 100: ffmpeg mit ABR, eigener Screenshot in Anlehnung an [147]

In einem ersten Schritt wird der eingehende RTMP-Stream, der als Input (*-i*) definiert ist, in drei verschiedene Versionen skaliert. Dabei wird das Eingangssignal zunächst in die drei Ausgangssignale *v1*, *v2* und *v3* aufgeteilt. Diese dienen im nächsten Schritt dem Input für eine *ffmpeg*-

Skalierungsfunktion, die jeweils das Eingangssignal neu skaliert [147]. Dabei enthält die erste Version die Originalauflösung, die zweite Variante eine Auflösung von 1280 x 720 Pixeln und die dritte Version eine Auflösung von 640 x 360 Pixeln. Die Output-Signale der Skalierungsfunktion werden mit *v1out*, *v2out* und *v3out* bezeichnet.

Im nächsten Schritt können die drei Signale, die in den Variablen *v1out*, *v2out* und *v3out* gespeichert sind, transkodiert werden. Dazu wird für jedes Signal der libx264-Algorithmus ausgewählt und jeweils die gewünschte Bitrate definiert. Zudem werden die bereits beschriebenen Optionen *preset*, *sc_threshold*, *keyint_min* und *g* gesetzt.

Im letzten Schritt können die HLS-Playlisten definiert werden. Dazu werden auch die Segmentlänge mit einer Sekunde und die Playlist-Länge mit 20 Segmenten angegeben und es wird der Dateiname der Segmente definiert. Dabei wird für jede HLS-Variante ein Ordner angelegt und es werden jeweils die dazugehörige Playliste und Segmente darin abgelegt. Im Kommando geschieht dies über den Platzhalter *%v*, was dem Index der Version entspricht. Außerdem wird mit *-master_pl_name* eine Masterplayliste definiert, die die URLs aller verfügbaren Playlisten enthält.

Wird das Skript gestartet, nimmt es den vom anderen Skript gesendeten RTMP-Stream entgegen und generiert drei Ordner für die unterschiedlichen Auflösungen und die Masterplaylist.

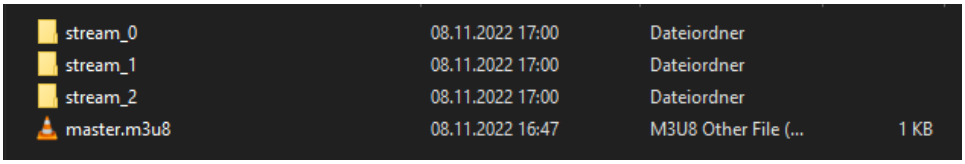

Abbildung 101: generierte Dateien mit ffmpeg-ABR-Streaming, eigener Screenshot

In jedem der drei Ordner werden schließlich die Segmente in den entsprechenden Auflösungen und eine individuelle Playliste abgelegt. Die Masterplaylist hingegen enthält die Dateipfade zu jeder Playliste und die entsprechenden Informationen zu Bandbreite, Auflösung und verfügbaren Codecs.

```
#EXTM3U
#EXT-X-VERSION:3
#EXT-X-STREAM-INF:BANDWIDTH=5500000,RESOLUTION=1920x1080,CODECS="avc1.640028"
stream_0/stream.m3u8

#EXT-X-STREAM-INF:BANDWIDTH=3300000,RESOLUTION=1280x720,CODECS="avc1.64001f"
stream_1/stream.m3u8

#EXT-X-STREAM-INF:BANDWIDTH=1100000,RESOLUTION=640x360,CODECS="avc1.64001e"
stream_2/stream.m3u8
```

Abbildung 102: Masterplayliste, eigener Screenshot

Beginnt ein Client, den Stream abzurufen, wird im ersten Schritt die Masterplaylist heruntergeladen. Damit erhält der Client Einsicht in die verfügbaren Qualitätsstufen der Stream-Segmente. Basierend auf der aktuellen Bandbreite des Clients entscheidet sich dieser schließlich für eine Variante und lädt die

Playlist und im Anschluss die Segmente der entsprechenden Variante herunter. Ändern sich die Netzwerkbedingungen, kann der Client auf eine andere Variante zurückgreifen.

Dies lässt sich mit den Netzwerkeinstellungen im Browser testen, in diesem Fall mit Firefox. Um das adaptive Bitrate-Streaming zu testen, wird im ersten Schritt der Webserver *nginx* gestartet und das bereits beschriebene Skript (siehe 8.2.1) zur Aufnahme des Bildschirms auf dem Windows-Rechner gestartet. Im Anschluss wird das in Abbildung 100 dargestellte Skript zum ABR-Streaming auf dem *nginx*-Server abgelegt und aufgerufen. Nun beginnt *ffmpeg* damit, den eingehenden RTMP-Stream in HLS-Segmente der definierten Auflösungen in den entsprechenden Ordnern auf dem Server abzulegen. Danach kann die HTML-Testseite aufgerufen und der Videoplayer gestartet werden.

Ein Blick in die Netzwerkanalyse der Entwicklereinstellungen des Browsers bestätigt, dass der Browser im ersten Schritt die Masterplaylist abruft, um Einsicht in die verfügbaren Segment-Varianten zu erhalten.

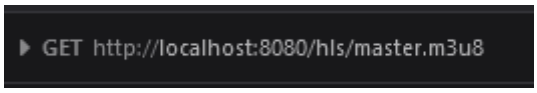

Abbildung 103: Abrufen der Masterplayliste, eigener Screenshot

Sobald der Client die Masterplaylist erhalten hat, kann die Playliste zur für diesen passenden Auflösung heruntergeladen werden. Der Client beginnt dann, die Segmente herunterzuladen und abzuspielen. Die Netzwerkanalyse gibt Auskunft darüber, welche Auflösung gewählt wird. Bei guter Netzwerkverbindung ohne manuelle Drosselung wird, wie erwartet, die bestmögliche Qualität, die in diesem Fall im Ordner *stream_0* abgelegt ist, heruntergeladen. Dies entspricht der Originalauflösung.

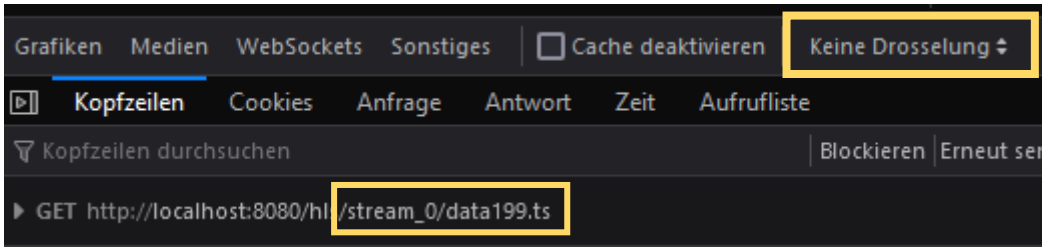

Abbildung 104: Abrufen eines Segments ohne Netzwerkdrosselung, eigener Screenshot

Zu Testzwecken kann die Netzwerkverbindung in der Netzwerkanalyse der Entwicklungseinstellungen manuell geändert werden. Um zu beobachten, wie der Client sich verhält, wird diese auf reguläres 3G gesetzt. Wie die folgende Abbildung zeigt, wird nun die Playlist der schlechteren Qualität im Ordner *stream_2* abgerufen und es wird damit begonnen, die Segmente in dieser Auflösung herunterzuladen.

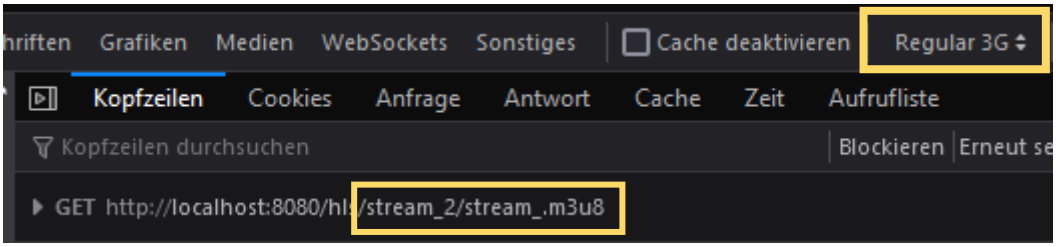

Abbildung 105: Abrufen der Playlist mit geringerer Qualität, eigener Screenshot

Dies demonstriert, wie ein Client, in Abhängigkeit von der für ihn verfügbaren Bandbreite, die jeweils passende Auflösung des Streams abspielen kann.

Zusammenfassend lässt sich sagen, dass das Streaming mit adaptiver Bitrate auch mit *ffmpeg* umgesetzt werden kann, um eine variable Auflösung beziehungsweise Bitrate zu bieten. Um das adaptive Bitrate-Streaming für den Prototypen einzusetzen, muss jedoch beachtet werden, dass es durch die erweiterte Architektur des Prototyps (siehe 10.) nötig ist, jedes Segment in jeder verfügbaren Auflösung von der implementierten C#-Rest-API abzurufen. Demnach müsste die C#-Rest-API um weitere Endpunkte für weitere Auflösungen erweitert werden. Hinzu kommt, dass die Latenz durch die ABR-Funktionalität zunehmen würde, da jedes Videosegment durch die verschiedenen Auflösungen mehrmals von den TBM-Rechnern in die Cloud übertragen werden müsste, was den Datenverkehr im Tunnel-Netzwerk stark erhöhen würde.

Demnach lässt sich sagen, dass das adaptive Bitrate-Streaming für dieses Projekt zunächst nicht relevant ist und daher auch nicht umgesetzt wurde. Der Lösungsansatz ist jedoch erweiterbar, sollte dies in der Zukunft für das Videostreaming auf *CONNECTED* erforderlich werden.

12.2. Latenz-Messung

In diesem Abschnitt geht es um die Messung der Latenz, also der Verzögerungszeit zwischen der eigentlichen Bildschirmaufnahme und der Wiedergabe des Streams im Videoplayer. Um dies durchzuführen, wurden mithilfe von Open Broadcast Studio (siehe 6.1.1.) Bildschirmaufnahmen des zu übertragenden Bildschirms und des Videoplayers mit der Wiedergabe des aufgenommenen Videostreams gemacht. Dabei wurde eine Verbindung zum entfernten Rechner per TeamViewer hergestellt, dessen Bildschirm aufgenommen wird. In Open Broadcast Studio wurden dann sowohl das TeamViewer-Fenster als auch das Browser-Fenster mit Videoplayer nebeneinander platziert und die Aufnahme wurde gestartet. Während der Aufnahme wurde schließlich ein Fenster auf dem aufgenommenen Bildschirm minimiert und ermittelt, wie viel Zeit zwischen der Minimierung des Fensters und der Wiedergabe derselben Aktion im Stream verging, der im Browser dargestellt wird.

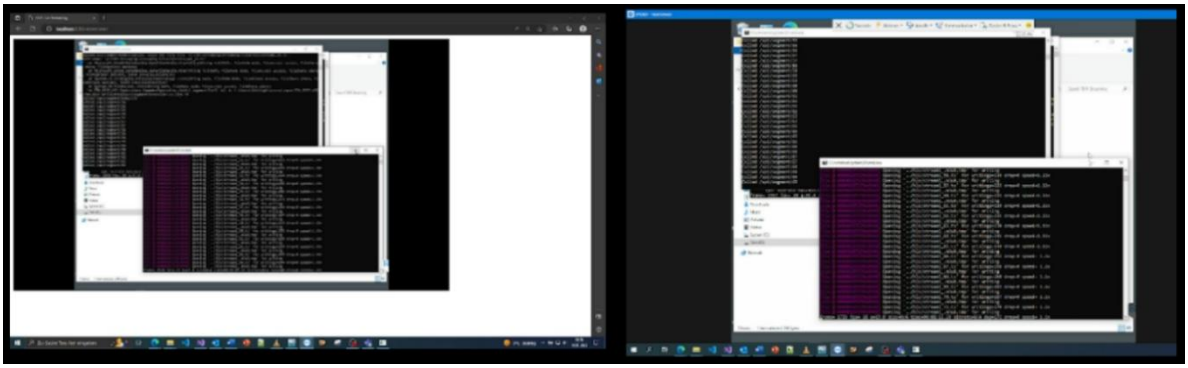

Abbildung 106: Vergleich aufgenommener Bildschirm (rechts) und Wiedergabe des Streams (links), eigener Screenshot

Nach der Aufnahme konnte diese im VLC Media Player abgespielt und die Latenz über die Zeitleiste bestimmt werden. Dazu wurden die Zeit, zu der das Fenster auf dem aufgenommenen Bildschirm (TeamViewer-Fenster) geschlossen wurde, sowie die Zeit derselben Aktion im dargestellten Stream notiert. Da der VLC Media Player standardmäßig lediglich die vergangenen Sekunden anzeigt, wurde eine Extension installiert, die zusätzlich die Millisekunden anzeigt [148]. Dadurch konnten die Zeitpunkte genauer abgelesen werden.

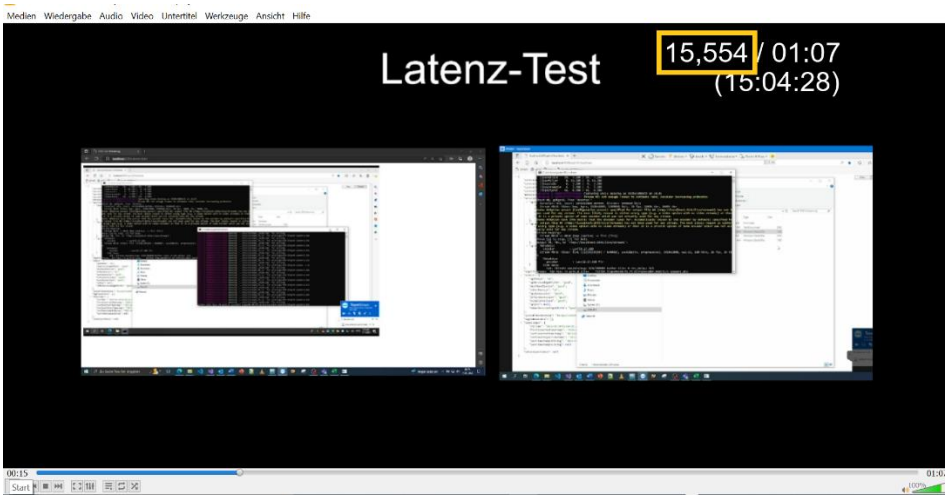

Abbildung 107: Latenz-Test mit Zeitangabe in Millisekunden, eigener Screenshot

So wurde in Abbildung 107 zum Beispiel das vordere Fenster (Kommandozeile) bereits geschlossen. Der Zeitpunkt, zu dem dies geschah, konnte über die installierte Anzeige in der rechten oberen Ecke (siehe gelbe Markierung) abgelesen werden. Danach wurde beobachtet, wann die Minimierung desselben Fensters im Stream (linke Seite in Abbildung 107) wiedergegeben wurde. Auch dieser Zeitpunkt wurde notiert. Im nächsten Schritt wurde die Differenz der notierten Zeitwerte berechnet, um die Latenz zu erhalten.

Für diesen Test wurden mehrere solcher Aufnahmen über mehrere Tage und zu verschiedenen Tageszeiten durchgeführt, um im Anschluss die durchschnittliche Latenz der implementierten Streaming-Lösung zu ermitteln.

Unterschieden wurden zwei verschiedene Test-Szenarien: dem Streaming auf einem lokalen Rechner ohne Rest-API und Rest-Client (1) und dem Streaming auf dem Testrechner (2).

Beim ersten Test-Szenario (1) wurden alle Komponenten auf demselben lokalen Rechner gestartet. Das bedeutet, dass die Bildschirmaufnahme und die Wiedergabe auf demselben Rechner durchgeführt wurden. Dabei wurde auf die Rest-API und den Rest-Client verzichtet, so wie es ursprünglich gedacht war (siehe 8.). Dabei wurden die Segmente direkt auf dem lokal installierten Webserver abgelegt und schließlich im aufgerufenen Videoplayer im Browser abgespielt.

Das zweite Test-Szenario (2) entspricht der neuen Architektur, die in Abschnitt 10. beschrieben wurde. Das bedeutet, dass die C#-Rest-API und die zwei *ffmpeg*-Skripte auf einem Testrechner gestartet wurden und der C#-Rest-Client auf einem lokalen Rechner die Segmente abruft und auf dem Webserver ablegt. Der Videoplayer wurde dann ebenso lokal aufgerufen, um den Stream abzuspielen.

Zur Latenzmessung wurden für jeden Test-Aufbau jeweils 16 Aufnahmen im Zeitraum vom 10.01.2023 bis zum 16.01.2023 zu unterschiedlichen Uhrzeiten gemacht. Dabei wurden, wie zuvor beschrieben wurde, jeweils die entsprechenden Zeitpunkte im VLC Media Player abgelesen und die Differenz der Werte wurde berechnet, um für jede Aufnahme die Latenz zu erhalten. Im Anschluss wurde für jeden

Test-Aufbau die durchschnittlich erreichte Latenz berechnet. Die Tabellen mit allen gemessenen Daten zur Berechnung der Latenz sind dem Anhang E zu entnehmen.

Folgende Tabelle zeigt eine Übersicht der minimal, maximal und durchschnittlich gemessenen Latenzen für jedes Test-Szenario:

Testaufbau	Min. (in s)	Max. (in s)	Durchschnittliche Latenz (in s)
(1) Lokal (ohne Rest-API & -Client)	3,75	5,00	4,33
(2) Testrechner (mit Rest-API & -Client)	5,25	7,75	7,10

Tabelle 4: Durchschnittlich erreichte Latenzen, eigene Darstellung

Die Tabelle 4 zeigt, dass der lokale Test-Aufbau ohne Rest-API und -Client (1) eine durchschnittliche Latenz von etwa 4,33 Sekunden erzielt. Die minimal gemessene Latenz innerhalb des Testzeitraums lag bei 3,75 Sekunden und die maximale Latenz bei 5,00 Sekunden.

Im Gegensatz dazu konnte mit dem zweiten Testszenario (2) durchschnittlich eine Verzögerungszeit von etwa 7,10 Sekunden erreicht werden. Die minimale Latenz lag dabei bei 5,25 Sekunden und die maximale Latenz bei 7,75 Sekunden.

Die Latenz ist demnach beim zweiten Testaufbau im Durchschnitt 2,77 Sekunden höher als beim lokalen Szenario ohne Rest-API und -Client. Der Grund dafür ist einerseits der zusätzliche Übertragungsschritt der Videosegmente zwischen der Rest-API und dem -Client. Außerdem trägt die räumliche Distanz zwischen dem Testrechner und dem lokalen Rechner dazu bei, dass die Latenz hier höher ist als beim ersten Test-Aufbau. Die räumliche Distanz ist jedoch nicht zu vermeiden, sondern im produktiven Gebrauch der Lösung sogar noch höher, da die Tunnelbohrmaschinen und somit die Rechner, deren Bildschirme aufgenommen werden, auf der gesamten Welt verteilt sind.

Laut GCore [59] liegt die allgemeine durchschnittliche Latenz beim Streaming bei 30 bis 40 Sekunden. Auch wenn die in dieser Thesis ermittelte Streaming-Lösung noch kein Echtzeitstreaming bietet, lässt sich festhalten, dass eine Latenz von unter acht Sekunden dennoch zufriedenstellend ist, insbesondere aufgrund der komplexen Systemarchitektur des Prototyps und des zusätzlichen Übertragungsschritts der Videosegmente zwischen API und Client.

13. Ausblick

In diesem Abschnitt geht es um die Zukunftsfähigkeit des implementierten Prototyps. Dabei soll ein Blick auf die Wartbarkeit der Lösung und mögliche Weiterentwicklungen geworfen werden, um den Lösungsansatz in der Zukunft zu optimieren.

Generell lässt sich sagen, dass die Architektur des Lösungsansatzes zwar komplex ist, aber insbesondere durch die Automatisierung (siehe Kapitel 11.) in der Zukunft wartungsarm sein wird. Das bedeutet, dass mögliche Fehler automatisch erkannt werden, beziehungsweise durch den automatischen Neustart der entsprechenden Komponenten Ausfallzeiten minimiert werden können, sodass die Lösung langfristig stabil laufen kann. Da die Automatisierung noch nicht praktisch umgesetzt wurde, ist es empfehlenswert, dies in der Zukunft durchzuführen, sodass die Streaming-Lösung mit wenig Aufwand automatisch auf eine Vielzahl von Maschinen angewendet werden und mit möglichst geringen Ausfallzeiten laufen kann.

Ein weiterer Vorteil der Lösung ist, dass die entwickelte Software-Architektur auch für das Streaming von Videokamerasignalen eingesetzt werden kann. Sicherheitskameras, die sich zum Teil auf den Tunnelbohrmaschinen beziehungsweise auf den entsprechenden Baustellen befinden, liefern Signale, die auf einem Rekorder aufgenommen werden. Daher ist es möglich, diese in der Zukunft vom Rekorder abzugreifen und mit den entsprechenden Komponenten des Prototyps zu segmentieren und als Stream zur Verfügung zu stellen. Demnach ist es möglich, in der Zukunft auch Videokamerasignale der Sicherheitskameras auf Baustellen in *CONNECTED* als Videostream zu integrieren.

Außerdem soll das Videostreaming in der Zukunft lediglich verfügbar sein, wenn die Authentifizierung und die Autorisierung es erlauben. Dazu gibt es bereits eine Lösung, die integriert werden kann. Der Prototyp, der im Rahmen dieses Projektes entwickelt wurde, ist noch nicht an diese angebunden, da das Videostreaming noch nicht für Kunden beziehungsweise ein laufendes Projekt verfügbar ist. Dennoch kann dies in der Zukunft umgesetzt werden, um zu verhindern, dass unberechtigte Personen Zugriff auf das Videostreaming erhalten.

In weiterführender Forschung könnten außerdem Möglichkeiten ermittelt werden, die Latenz zur Übertragung noch weiter zu reduzieren, um noch näher an eine Echtzeitübertragung zu gelangen. Dabei könnte die Architektur eventuell vereinfacht werden oder es könnten weitere Komprimierungsverfahren getestet werden.

14. Zusammenfassung und Fazit

Zum Abschluss werden an dieser Stelle die wesentlichen Aspekte des Projektes zusammengefasst und das erzielte Ergebnis wird evaluiert.

Zu Beginn dieser Thesis wurden das Unternehmen, die Kundenplattform Herrenknecht *CONNECTED*, sowie die Projektanforderungen vorgestellt. Im Anschluss ging es um die Datenübertragung im Internet und das Media-Streaming, um ein Grundverständnis für das Thema des Projekts aufzubauen. Es lässt sich festhalten, dass Media-Streaming gegenwärtig eine große Bedeutung hat und verschiedene Protokolle zum Einsatz kommen können, wenn es darum geht, einen Videostream für Clients verfügbar zu machen.

Im Anschluss wurde eine Marktübersicht verschiedener Streaming-Komponenten gegeben. Dies diente als Basis für das darauffolgende Kapitel, in dem verschiedene Lösungsansätze recherchiert und verglichen wurden. Dabei wurden auch diverse Vergleichskriterien anhand der Projektanforderungen bestimmt, um die ermittelten Ansätze untereinander abzuwägen und den für das Projekt geeignetsten Ansatz zu wählen.

Daraufhin ging es um die Implementierung des gewählten Lösungsansatzes. Dabei wurde ein erster Prototyp lokal auf einem Rechner entwickelt, der den Bildschirminhalt des jeweiligen Rechners mittels *ffmpeg* aufnimmt und als RTMP-Stream an einen RTMP-Server sendet. Der RTMP-Stream wird im nächsten Schritt von *ffmpeg* entgegengenommen und in einzelne Videosegmente zerlegt, die mithilfe der dabei generierten Playlist-Datei von Clients abgerufen und als Stream innerhalb eines Videoplayers wiedergegeben werden können.

Im nächsten Schritt ging es darum, das *ffmpeg*-Skript zur Bildschirmaufnahme auf einen Testrechner zu übertragen, um die Software-Verteilung der gewählten Architektur zu testen. Dabei konnte keine Verbindung zwischen dem Testrechner und dem Container in der Cloud geschaffen werden, da RTMP die Übertragung des Streams per HTTP nicht unterstützt. Deshalb wurde die Softwarearchitektur im nächsten Kapitel angepasst. Dabei wurde sie um eine C#-Rest-API auf dem TBM-Rechner und einen C#-Rest-Client auf dem Server erweitert, um die erstellten Videosegmente per API vom TBM-Rechner auf den Server zu übertragen. Kapitel 10 umfasst die Entwicklung der Rest-API und des Rest-Clients in C#, um schließlich einen funktionsfähigen Prototyp zu erhalten.

Danach wurden Möglichkeiten vorgestellt, die in der Zukunft genutzt werden können, um die Softwareinstallation zu automatisieren. Dabei wurde die Kubernetes-Integration erläutert und es wurden zwei Software-Tools vorgestellt, die bei der Herrenknecht AG eingesetzt werden, um Software automatisiert auf TBM-Rechner zu übertragen und zu starten.

In Kapitel 12 wurden verschiedene *ffmpeg*-Optionen getestet, um das adaptive Bitrate-Streaming zu testen. Allerdings wurde für den Lösungsansatz darauf verzichtet, um den Kompressionsaufwand und

damit die Verzögerungszeit minimal zu halten. Außerdem wurde die durchschnittlich erreichte Latenz bei zwei unterschiedlichen Testszenarien gemessen. Für den Prototyp konnte eine durchschnittliche Latenz von etwa 7,10 Sekunden ermittelt werden.

Zusammenfassend lässt sich sagen, dass das Ziel dieser Masterarbeit, einen Lösungsansatz zu ermitteln und ein Prototyp für das Streaming auf *CONNECTED* zu implementieren, erreicht und die wichtigsten Projektanforderungen damit erfüllt werden konnten. Dennoch wird das Ergebnis dieses Projektes im Folgenden kurz bewertet.

Das Videostreaming für *CONNECTED* konnte prototypisch in einer Testumgebung umgesetzt werden. Dazu kamen zwei verschiedene Streaming-Technologien zum Einsatz: RTMP für den Streaming-*Ingest* und HLS für die Segmentierung des Videos in einzelne, abrufbare Segmente.

Bei der Softwarearchitektur ergaben sich zunächst einige Herausforderungen, da der Zugriff auf die TBM-Rechner stark eingeschränkt ist. Grund dafür ist die Tatsache, dass es sich bei den Bildschirminhalten auf den Tunnelbohrmaschinen um sensible Daten handelt, die Auskunft über den aktuellen Stand der Maschine geben. Daher war es wichtig, sicherzustellen, dass Dritte keinen Zugriff auf die Rechner erhalten und möglicherweise für die Maschinensteuerung wichtige Parameter ändern können. Daraus ergab sich zunächst die Herausforderung, eine Möglichkeit zu finden, die Bildschirmaufnahme der TBM-Rechner sicher auf den Server zu übertragen. Letztlich war es jedoch möglich, den Videostream bereits auf den TBM-Rechnern zu segmentieren und per API an einen Server zu liefern. Somit konnte die Anforderung erfüllt werden, dass die Daten nur einmal von den TBM-Rechnern in die Cloud übertragen und von dort auf *CONNECTED* bereitgestellt werden.

Ein weiteres Ziel des Projekts bestand darin, die Latenz geringzuhalten und die Übertragung in nahezu Echtzeit zu ermöglichen. Mit der Kombination aus RTMP und HLS und der gewählten Softwarearchitektur war es möglich, eine Latenz von durchschnittlich etwa 7,10 Sekunden zu erreichen, was in Anbetracht der komplexen Architektur des Prototyps zufriedenstellend ist.

Optimal sollte eine Lösung gewählt werden, die keine zusätzliche Software auf den TBM-Rechnern benötigt. Allerdings war es nicht möglich, eine funktionsfähige Lösung zu finden, bei der die bereits installierten VNC-Server auf den TBM-Rechnern genutzt werden. Aus diesem Grund war es notwendig, auf andere Software zurückzuzugreifen, um eine kompatible Lösung zu finden. Daraus ergab sich eine komplexere Lösung, als ursprünglich angenommen wurde.

Dennoch konnte eine Lösung gefunden werden, die nicht nur mit allen TBM-Rechnern kompatibel ist, sondern in der Zukunft auch für Kamerasignale eingesetzt werden kann. Außerdem ist der Lösungsansatz skalierbar und kann in zukünftig auf eine große Anzahl an Maschinen angewendet werden. Demnach kann es Kunden und sonstigen Projektbeteiligten ermöglicht werden, die Navigations- und Steuerungsanzeigen ihrer Tunnelbohrmaschinen in nahezu Echtzeit auf *CONNECTED* anzusehen und dadurch einen Überblick über den aktuellen Stand ihrer Maschinen zu erhalten.

15. Literaturverzeichnis

[1] T. Wang, „The Construction of Online Video Resource Library with Streaming Media,“ in *Knowledge Discovery and Data Mining*, Berlin, Heidelberg, Springer, 2012, p. 701–708.

[2] J. L. Wenjun, „Real Time Multimedia,“ in *Encyclopedia of Multimedia*, Boston, MA, Springer, 2008, p. 757–762.

[3] T. Ruether, „History of Streaming Media [Infographic],“ 25. 04. 2022. [Online]. Available: https://www.wowza.com/blog/history-of-streaming-media. [Zugriff am 05. 09. 2022].

[4] Herrenknecht, „Konzern,“ 2022. [Online]. Available: https://www.herrenknecht.com/de/unternehmen/konzern/. [Zugriff am 02. 09. 2022].

[5] Herrenknecht, „Unternehmen,“ 2022. [Online]. Available: https://www.herrenknecht.com/de/produkte/tunnelling/. [Zugriff am 02. 09. 2022].

[6] Herrenknecht, „GIGANTISCHE STRASSENTUNNEL IN SHANGHAI,“ Herrenknecht AG, [Online]. Available: https://www.herrenknecht.com/de/referenzen/referenzendetail/gigantische-strassentunnel-in-shanghai/. [Zugriff am 23. 01. 2023].

[7] Herrenknecht, „HONG KONG TUEN MUN – CHEK LAP KOK LINK,“ Herrenknecht AG, 2022. [Online]. Available: https://www.herrenknecht.com/de/referenzen/referenzendetail/hong-kong-tuen-mun-chek-lap-kok-link/. [Zugriff am 07. 11. 2022].

[8] H. AG, „Unternehmen,“ Herrenknecht AG, 2023. [Online]. Available: https://www.herrenknecht.com/de/unternehmen/. [Zugriff am 19. 01. 2023].

[9] Herrenknecht, „Tunneling,“ Herrenknecht AG, 2023. [Online]. Available: https://www.herrenknecht.com/de/produkte/tunnelling/. [Zugriff am 13. 01. 2023].

[10] Herrenknecht, „EPB-SCHILD,“ Herrenknecht AG, 2022. [Online]. Available: https://www.herrenknecht.com/de/produkte/productdetail/epb-schild/. [Zugriff am 07. 11. 2022].

[11] Herrenknecht, „Ver- und Entsorgungssysteme,“ Herrenknecht AG, 2023. [Online]. Available: https://www.herrenknecht.com/de/produkte/tunnelling/ver-und-entsorgungssysteme/. [Zugriff am 13. 01. 2023].

[12] Herrenknecht, „Mining,“ 2022. [Online]. Available: https://www.herrenknecht.com/de/produkte/mining/. [Zugriff am 02. 09. 2022].

[13] Herrenknecht, „Exploration,“ 2022. [Online]. Available: https://www.herrenknecht.com/de/produkte/exploration/. [Zugriff am 02. 09. 2022].

[14] Herrenknecht, „Zusatzequipment,“ 2022. [Online]. Available: https://www.herrenknecht.com/de/produkte/zusatzequipment/. [Zugriff am 02. 09. 2022].

[15] Herrenknecht, „Herrenknecht CONNECTED,“ Herrenknecht AG, 2022. [Online]. Available: https://connected.herrenknecht.com/connected/. [Zugriff am 2022].

[16] Herrenkencht, „DEMO MT," Herrenknecht AG, 2022. [Online]. Available: https://connected.herrenknecht.com/connected/fleet?machine=demomt&selectionLevel=Machine. [Zugriff am 18. 11. 2022].

[17] Herrenknecht, „NAVI," Herrenknecht AG, 2023. [Online]. Available: https://connected.herrenknecht.com/connected/demoepbshield/visualization?streamName=navi. [Zugriff am 13. 01. 2023].

[18] Herrenknecht, „VISU," Herrenknecht AG, 2023. [Online]. Available: https://connected.herrenknecht.com/connected/demomt/visualization?streamName=visu. [Zugriff am 13. 01. 2023].

[19] V. Wang, F. Salim und P. Moskovits, „VNC with the Remote Framebuffer Protocol," in *The Definitive Guide to HTML5 WebSocket*, Berkeley, CA, Apress, 2013, pp. 109-127.

[20] T. Richardson, „The RFB Protocol," RealVNC Ltd, 2003.

[21] P.-B. Bök, M. Müller, A. Noack und D. Behnke, „Einführung," in *Computernetze und Internet of Things*, Wiesbaden, Springer Vieweg, 2020, pp. 7-56.

[22] J. Scherff, „Einführung," in *Grundkurs Computernetze*, Vieweg+Teubner, 2006, pp. 1-25.

[23] C. Baun, Computernetze kompakt, Heidelberg: Springer Vieweg Berlin, 2020.

[24] A. Luntovskyy und D. Gütter , „ISO-Architektur," in *Moderne Rechnernetze*, Wiesbaden, Springer Vieweg, 2020, p. 49–62.

[25] P. Solutions, „What is the OSI model?," Polymer Solutions, 20. 03. 2021. [Online]. Available: https://www.polymerhq.io/blog/cloud-security/post-what-is-the-osi-model/. [Zugriff am 21. 11. 2022].

[26] J. Evans, „The OSI model doesn't map well to TCP/IP," 2021. [Online]. Available: https://jvns.ca/blog/2021/05/11/what-s-the-osi-model-/. [Zugriff am 21. 11. 2022].

[27] N. Lippis, „The OSI Model Is Dead," 451 Alliance, 15. 04. 2019. [Online]. Available: https://blog.451alliance.com/the-osi-model-is-dead/. [Zugriff am 21. 11. 2022].

[28] S. Mishra, „The TCP/IP Model: An Overview," *International Journal of Linguistics & Computing Research,* 12. 2017.

[29] A. Keller, „OSI-Layer und Protokolle," in *Breitbandkabel und Zugangsnetze*, Berlin, Heidelberg, Springer, 2011, pp. 289-326.

[30] Y. Sani, A. Mauthe und C. Edwards, „Adaptive Bitrate Selection: A Survey," *IEEE COMMUNICATIONS SURVEYS & TUTORIALS,* Bd. 19, Nr. 4, pp. 2985-3014, 2017.

[31] A. Edney, „Media Streaming," in *Windows Home Server User's Guide*, Apress, 2007, pp. 213-227.

[32] B. Jedari, G. Premsankar, G. Illahi, M. Di Francesco, A. Mehrabi und A. Ylä-Jääski, „Video Caching, Analytics, and Delivery at the Wireless Edge: A Survey and Future Directions," *IEEE Communications Surveys & Tutorials,* Bd. 23, Nr. 1, pp. 431-471, 09. 11. 2020.

[33] J. Krikke, „Streaming video transforms the media industry," *IEEE Computer Graphics and Applications,* pp. 6-12, 12. 07. 2004.

[34] W. Hürst, T. Lauer und R. Müller, „Enhanced Interaction for Streaming Media," in *E-Business and Telecommunication Networks. ICETE 2006. Communications in Computer and Information Science,* 2008.

[35] R. Awati, „streaming media," 08. 2022. [Online]. Available: https://www.techtarget.com/whatis/definition/streaming-media. [Zugriff am 05. 09. 2022].

[36] L. Chen, Y. Zhou und D. M. Chiu, „Smart Progressive Downloading," 2014.

[37] R. Pereira und E. G. Pereira, „Dynamic Adaptive Streaming over HTTP and Progressive Download: Comparative Considerations," in *2014 28th International Conference on Advanced Information Networking and Applications Workshops,* Victoria, BC, Canada, 2014.

[38] J. W. Smith, „The Best Live Streaming Encoders (2022)," 28. 12. 2021. [Online]. Available: https://www.joelwsmith.com/live-streaming-encoders. [Zugriff am 18. 10. 2022].

[39] V. V. Pirozhenko und V. D. Grigoriev, „Video Stream Processing and Compression with Codec Choice Ability," in *2020 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus),* St. Petersburg and Moscow, Russia, 2020.

[40] E. Krings, „What is RTMP Ingest and Why is it Important for Live Streaming?," 17. 06. 2022. [Online]. Available: https://www.dacast.com/blog/rtmp-ingest/. [Zugriff am 20. 10. 2022].

[41] E. Krings, „The Definitive Guide to Video Streaming Technology in 2022," dacast, 27. 06. 2022. [Online]. Available: https://www.dacast.com/blog/video-streaming-technology/#:~:text=The%20four%20major%20types%20of%20streaming%20technology%20include,roles%20that%20they%20play%20in%20online%20video%20streaming.. [Zugriff am 08. 09. 2022].

[42] U. Stanimirovic, „Optimizing for User Experience With Adaptive Bitrate Streaming," 02. 12. 2020. [Online]. Available: https://site.brid.tv/adaptive-bitrate-streaming/. [Zugriff am 09. 09. 2022].

[43] T. Kimura, T. Kimura, A. Matsumoto und K. Yamagishi, „Balancing Quality of Experience and Traffic Volume in Adaptive Bitrate Streaming," *IEEE Access,* pp. 15530 - 15547, 18. 01. 2021.

[44] A. Bybyk, „Low latency: What it is and how to get it," 14. 11. 2019. [Online]. Available: https://restream.io/blog/what-is-low-latency/#:~:text=Streaming%20protocols%20to%20deliver%20low-latency%20video%20streams%201,the%20streaming%20platform%20Mixer%2C%20owned%20by%20Microsoft.%20. [Zugriff am 15. 09. 2022].

[45] P. Chakraborty, S. Dev und R. H. Naganur, „Dynamic HTTP Live Streaming Method for Live Feeds," in *2015 International Conference on Computational Intelligence and Communication Networks (CICN),* Jabalpur, India, 2015.

[46] A. Fecheyr-Lippens, „A Review of HTTP Live Streaming," 2010.

[47] H. S. Spilker und T. Colbjørnsen, „The dimensions of streaming: toward a typology of an evolving concept," *Media, Culture & Society,* Bd. 42, Nr. 7-8, p. 1210–1225, 27. 02. 2020.

[48] A. Bybyk, „The history of live streaming,“ 02. 02. 2021. [Online]. Available: https://restream.io/blog/history-of-live-streaming/. [Zugriff am 08. 09. 2022].

[49] I. Sodagar, „The MPEG-DASH Standard for Multimedia Streaming Over the Internet,“ *IEEE MultiMedia* , Bd. 18, Nr. 4, pp. 62-67, 04. 2011.

[50] I. Santos-González, A. Rivero-García, J. Molina-Gil und P. Caballero-Gil, „Implementation and Analysis of Real-Time Streaming Protocols,“ Tenerife, Spain, 2017.

[51] T. Koistinen, „Protocol overview: RTP and RTCP,“ 1999.

[52] H. O. Inge, J. Ola und P. Andrew, „RTP-based broadcast streaming of high definition H.264/AVC video: An error robustness evaluation,“ *Journal of Zhejiang University-SCIENCE A*, pp. 19-26, 2006.

[53] A. Bychok, „Video streaming protocols explained: RTMP, WebRTC, FTL, SRT,“ 26. 05. 2020. [Online]. Available: https://restream.io/blog/streaming-protocols/#:~:text=Most-used%20streaming%20protocols%201%20RTMP%20%28Real%20Time%20Messaging,...%204%20SRT%20%28Secure%20Reliable%20Transport%29%3A%20UDP%20. [Zugriff am 11. 10. 2022].

[54] P. Henken, „Video Streaming Protocols - An Introduction,“ 19. 04. 2021. [Online]. Available: https://corp.kaltura.com/blog/video-streaming-protocols/. [Zugriff am 05. 09. 2022].

[55] iana, „Real-Time Transport Protocol (RTP) Parameters,“ 18. 02. 2022. [Online]. Available: https://www.iana.org/assignments/rtp-parameters/rtp-parameters.xhtml. [Zugriff am 27. 01. 2023].

[56] T. Ruether, „What Is WebRTC? (Update),“ 24. 10. 2022. [Online]. Available: https://www.wowza.com/blog/what-is-webrtc. [Zugriff am 04. 11. 2022].

[57] L. Lanka, „Streaming Protocols – Everything you need to know,“ 20. 08. 2021. [Online]. Available: https://www.muvi.com/blogs/streaming-protocols-everything-you-need-to-know.html. [Zugriff am 11. 11. 2022].

[58] T. Ruether, „MPEG-DASH: Dynamic Adaptive Streaming Over HTTP Explained (Update),“ 18. 04. 2022. [Online]. Available: https://www.wowza.com/blog/mpeg-dash-dynamic-adaptive-streaming-over-http. [Zugriff am 14. 11. 2022].

[59] Core, „CMAF technology. Low Latency Streaming,“ [Online]. Available: https://gcore.com/support/articles/4405119953297/#:~:text=Low%20Latency%20MPEG-DASH%20is%20a%20video%20delivery%20technology,to%204-5%20seconds.%20How%20Low%20Latency%20MPEG-DASH%20works. [Zugriff am 14. 11. 2022].

[60] T. Ruether, „RTMP Streaming: The Real-Time Messaging Protocol Explained (Update),“ 07. 11. 2022. [Online]. Available: https://www.wowza.com/blog/rtmp-streaming-real-time-messaging-protocol. [Zugriff am 27. 01. 2023].

[61] H. Duhamel, „What is RTMP? The Real-Time Messaging Protocol: What you Need to Know in 2022,“ 17. 08. 2022. [Online]. Available: https://www.dacast.com/blog/rtmp-real-time-messaging-protocol/. [Zugriff am 20. 10. 2022].

[62] Cloudflare, „Was ist HTTP-Livestreaming? | HLS Streaming,“ CLOUDFLARE, 2023. [Online]. Available: https://www.cloudflare.com/de-de/learning/video/what-is-http-live-streaming/. [Zugriff am 27. 01. 2023].

[63] T. C. Thang, Q.-D. Ho, J. W. Kang und A. T. Pham, „Adaptive Streaming of Audiovisual Content using MPEG DASH,“ *IEEE Transactions on Consumer Electronics,* Bd. 58, Nr. 1, pp. 78-85, 2012.

[64] B. Jansen, T. Goodwin, V. Gupta, F. Kuipers und G. Zussman, „Performance Evaluation of WebRTC-based Video,“ *ACM SIGMETRICS Performance Evaluation Review,* Bd. 45, Nr. 3, p. 56–68, 2017.

[65] M. Soni und B. S. Rajput , „Security and Performance Evaluations of QUIC Protocol,“ in *Data Science and Intelligent Applications*, Singapore, Springer, 2021, p. 457–462.

[66] T. Viernickel, A. Frömmgen, A. Rizk, B. Koldehofe und R. Steinmetz, „Multipath QUIC: A Deployable Multipath Transport Protocol,“ in *2018 IEEE International Conference on Communications (ICC)*, Kansas City, MO, USA, 2018.

[67] A. Luis und M. A. Patricio, „Scalable Streaming of JPEG 2000 Live Video Using RTP over UDP,“ in *International Symposium on Distributed Computing and Artificial Intelligence 2008 (DCAI 2008)*, Berlin, Heidelberg, 2008.

[68] 3CX, „RTP und SRTP – was steckt hinter den Protokollen?,“ 3CX, [Online]. Available: https://www.3cx.de/voip-sip/rtp/. [Zugriff am 27. 01. 2023].

[69] H. Schulzrinne, S. L. Casner, R. Frederick und V. Jacobson, „RTP: A Transport Protocol for Real-Time Applications,“ The Internet Society, 2003.

[70] X. Lei, X. Jiang und C. Wang, „Design and implementation of streaming media processing software based on RTMP,“ in *2012 5th International Congress on Image and Signal Processing*, 2012.

[71] Y. Liu, B. Du, S. Wang, H. Yang und X. Wang, „Design and Implementation of Performance Testing Utility for RTSP Streaming Media Server,“ in *2010 First International Conference on Pervasive Computing, Signal Processing and Applications*, Harbin, China, 2010.

[72] R. Pantos und W. May, „HTTP Live Streaming,“ 08. 2017. [Online]. Available: https://www.rfc-editor.org/rfc/rfc8216. [Zugriff am 20. 10. 2022].

[73] K. Menon, „HTTP Live Streaming,“ [Online]. Available: https://startupdope.com/http-live-streaming/. [Zugriff am 09. 09. 2022].

[74] S. Wei und V. Swaminathan, „Low Latency Live Video Streaming over HTTP 2.0,“ in *NOSSDAV '14: Proceedings of Network and Operating System Support on Digital Audio and Video Workshop*, Singapore, Singapore, 2014.

[75] B. Sredojev, D. Samardzija und D. Posa, „WebRTC technology overview and signaling solution design and implementation,“ in *2015 38th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, Opatija, Croatia, 2015.

[76] S. Dutton, „Getting Started with WebRTC,“ 2012. [Online]. Available: https://web.dev/webrtc-basics/. [Zugriff am 02. 11. 2022].

[77] G. Carlucci, L. De Cicco und S. Mascolo, „HTTP over UDP: an Experimental Investigation of QUIC," in *SAC '15: Proceedings of the 30th Annual ACM Symposium on Applied Computing*, Salamanca, Spain, 2015.

[78] S. Ahmad und M. J. Arshad, „Enhancing Fast TCP's Performance Using Single TCP Connection for Parallel Traffic Flows to Prevent Head-of-Line Blocking," *IEEE Access*, Bd. 7, pp. 148152-148162, 09. 10. 2019.

[79] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar, J. Bailey, J. Dorfman, J. Roskind, J. Kulik, P. Westin, R. Tenneti, R. Shade, R. Hamilton, V. Vasiliev, W.-T. Chang und Z. Shi, „The QUIC Transport Protocol: Design and Internet-Scale Deployment," in *SIGCOMM '17: Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, Los Angeles, CA, USA, 2017.

[80] F. E. Lagarde, „Comparing the 12 Best Live Streaming Encoder Software + Hardware Solutions [Updated for 2022]," 26. 05. 2022. [Online]. Available: https://www.dacast.com/blog/live-stream-encoding-software/. [Zugriff am 18. 10. 2022].

[81] O. Studio, „OBS Studio," 2022. [Online]. Available: https://obsproject.com/. [Zugriff am 18. 10. 2022].

[82] E. Trainor-Fogleman, „13 trusted platforms that offer low-latency video streaming," 11. 07. 2022. [Online]. Available: https://www.evercast.us/blog/low-latency-video-streaming#:~:text=Average%20latency%3A%20Normal%20latency%20for%20OBS%20is%20around,can%20be%20as%20low%20as%20around%201-2%20seconds.. [Zugriff am 18. 10. 2022].

[83] S. Kenlon, „Open source live streaming with Open Broadcaster Software," 29. 04. 2020. [Online]. Available: https://opensource.com/article/20/4/open-source-live-stream. [Zugriff am 18. 10. 2022].

[84] F. developers, „ffmpeg Documentation," 2022. [Online]. Available: https://ffmpeg.org/ffmpeg.html. [Zugriff am 13. 10. 2022].

[85] F. developers, „About FFmpeg," 2022. [Online]. Available: https://ffmpeg.org/about.html. [Zugriff am 13. 10. 2022].

[86] J. Johnson, „12 Best Live Streaming Encoders for Content Creators in 2022," 04. 08. 2022. [Online]. Available: https://www.uscreen.tv/blog/streaming-encoder/#:~:text=12%20Best%20Live%20Streaming%20Encoders%201%201.%20OBS,5.%20Teradek%20...%206%206.%20LiveU%20Solo%20. [Zugriff am 02. 11. 2022].

[87] R. Team, „The 10 best live streaming encoders," 15. 09. 2022. [Online]. Available: https://restream.io/blog/live-streaming-encoder/#:~:text=%F0%9F%94%B4%205%20best%20software%20encoders%201%201.%20OBS,4%204.%20Wirecast%20...%205%205.%20VidBlasterX%20. [Zugriff am 18. 10. 2022].

[88] D. R. Bhojani, V. J. Dwivedi und R. M. Thanki, Hybrid Video Compression Standard, Singapore: Springer, 2020.

[89] N. Barman und M. G. Martini, „H.264/MPEG-AVC, H.265/MPEG-HEVC and VP9 codec comparison for live gaming video streaming," in *2017 Ninth International Conference on Quality of Multimedia Experience (QoMEX)*, Erfurt, Germany, 2017.

[90] M. Wilbert, „Streaming Codecs for Video and Audio: What Broadcasters Need to Know in 2022," 09. 08. 2022. [Online]. Available: https://www.dacast.com/blog/codec-basics-for-online-video-audio-and-live-streaming/. [Zugriff am 09. 11. 2022].

[91] D. Mukherjee, J. Bankoski, A. Grange, J. Han, J. Koleszar, P. Wilkins, Y. Xu und R. Bultje, „The latest open-source video codec VP9 – An overview and preliminary results," in *2013 Picture Coding Symposium (PCS)*, San Jose, CA, USA, 2013.

[92] E. Ann, „Top 12 Live-Streaming-Plattformen, die Sie nicht verpassen dürfen (Update 2020)," 08. 06. 2022. [Online]. Available: https://www.iskysoft.com/de/videobearbeitung/live-stream-platform.html. [Zugriff am 18. 10. 2022].

[93] I. Chan, „How To Make Your Own Live Streaming Server (DIY Guide)," uscreen, 14. 07. 2022. [Online]. Available: https://www.uscreen.tv/blog/make-your-own-live-streaming-server/. [Zugriff am 11. 11. 2022].

[94] B. Zolfaghari, G. Srivastava, S. Roy, H. R. Nemati, F. Afghah, T. Koshiba, A. Razi, K. Bibak, P. Mitra und B. K. Rai, „Content Delivery Networks: State of the Art, Trends, and Future Roadmap," *ACM Computing Surveys*, Bd. 53, Nr. 2, pp. 1-34, 17. 05. 2020.

[95] R. Pro, „RED5 MEDIA SERVER," 2020. [Online]. Available: https://www.red5pro.com/red5-media-server/. [Zugriff am 11. 11. 2022].

[96] A. M. S. Inc., „Ant Media," Ant Media Server Inc., 2022. [Online]. Available: https://antmedia.io/. [Zugriff am 11. 11. 2022].

[97] Adobe, „Adobe Flash Player EOL General Information Page," Adobe, 2022. [Online]. Available: https://www.adobe.com/products/flashplayer/end-of-life.html. [Zugriff am 14. 11. 2022].

[98] VideoLAN, „VLC media player," [Online]. Available: https://www.videolan.org/vlc/. [Zugriff am 14. 11. 2022].

[99] B. GmbH, „VLC Web Browser Plug-in," BurdaForward GmbH, 2022. [Online]. Available: https://www.chip.de/downloads/VLC-Web-Browser-Plug-in_79183431.html. [Zugriff am 14. 11. 2022].

[100] K. Spriestersbach, „Die besten HTML5-Video-Player," Webmasterpro, 21. 12. 2021. [Online]. Available: https://www.webmasterpro.de/coding/html/html5-video-player/. [Zugriff am 14. 11. 2022].

[101] JWP, „JWP," Longtail Ad Solutions, Inc., 2022. [Online]. Available: https://www.jwplayer.com/html5-video-player/. [Zugriff am 14. 11. 2022].

[102] StreamingVideoProvider, „THE BEST STREAMING VIDEO PLAYER FOR ANY WEBSITE," StreamingVideoProvider, 2022. [Online]. Available: https://www.streamingvideoprovider.com/streaming-video-player/. [Zugriff am 14. 11. 2022].

[103] G. Contributors, „noVNC," 2021. [Online]. Available: https://github.com/novnc/noVNC. [Zugriff am 03. 11. 2022].

[104] D. Nanni, „How to access VNC remote desktop in web browser,“ 23. 11. 2020. [Online]. Available: https://www.xmodulo.com/access-vnc-remote-desktop-web-browser.html. [Zugriff am 19. 09. 2022].

[105] Y. Shinyama, „vnc2flv,“ 2009. [Online]. Available: https://www.unixuser.org/~euske/python/vnc2flv/index.html. [Zugriff am 19. 12. 2022].

[106] M. E. Shacklett und S. Johnson, „WebRTC (Web Real-Time Communications),“ TechTarget, 03. 2022. [Online]. Available: https://www.computerweekly.com/de/definition/WebRTC-Web-Real-Time-Communications. [Zugriff am 15. 11. 2022].

[107] ffmpeg, „ffmpeg,“ [Online]. Available: https://ffmpeg.org/download.html#build-windows. [Zugriff am 07. 11. 2022].

[108] videohelp.com, „Software » Screen capture » UScreenCapture 2.0.18,“ [Online]. Available: https://www.videohelp.com/software/UScreenCapture. [Zugriff am 07. 11. 2022].

[109] D. Johnson, „Doug Johnson Productions,“ [Online]. Available: https://djp.li/rtmpstreaming. [Zugriff am 15. 11. 2022].

[110] ffmpeg, „DirectShow,“ 2021. [Online]. Available: https://trac.ffmpeg.org/wiki/DirectShow. [Zugriff am 08. 11. 2022].

[111] T. contributors, „H.264 Video Encoding Guide,“ 2022. [Online]. Available: https://trac.ffmpeg.org/wiki/Encode/H.264. [Zugriff am 13. 10. 2022].

[112] M. Riedl, „Using FFmpeg as a HLS streaming server (Part 2) – Enhanced HLS Segmentation,“ 24. 08. 2018. [Online]. Available: https://www.martin-riedl.de/2018/08/24/using-ffmpeg-as-a-hls-streaming-server-part-2/. [Zugriff am 08. 11. 2022].

[113] ITWissen.info, „group of picture (MPEG) (GOP),“ DATACOM Buchverlag GmbH, 30. 07. 2015. [Online]. Available: https://itwissen.info/group-of-picture-MPEG-GOP.html. [Zugriff am 13. 10. 2022].

[114] Á. Huszák und S. Imre, „Analysing GOP structure and packet loss effects on error propagation in MPEG-4 video streams,“ *2010 4th International Symposium on Communications, Control and Signal Processing (ISCCSP),* pp. 1-5, 2010.

[115] OTTVerse, „RTMP Streaming using FFmpeg Tutorial,“ 30. 10. 2021. [Online]. Available: https://ottverse.com/rtmp-streaming-using-ffmpeg-tutorial/. [Zugriff am 08. 11. 2022].

[116] I. Brightcove, „Getting Started,“ Brightcove, Inc., 2022. [Online]. Available: https://videojs.com/getting-started. [Zugriff am 13. 10. 2022].

[117] G. Contributors, „videojs / video.js,“ 2023. [Online]. Available: https://github.com/videojs/video.js/. [Zugriff am 11. 01. 2023].

[118] B. B. Rad, H. J. Bhatti und M. Ahmadi, „An Introduction to Docker and Analysis of its Performance,“ *IJCSNS International Journal of Computer Science and Network Security,* Bd. 17, Nr. 3, pp. 228-235, 2017.

[119] D. Inc., „Use containers to Build, Share and Run your applications,“ Docker Inc., 2022. [Online]. Available: https://www.docker.com/resources/what-container/. [Zugriff am 29. 11. 2022].

[120] tiangolo, „tiangolo/nginx-rtmp,“ Docker Inc., 2022. [Online]. Available: https://hub.docker.com/r/tiangolo/nginx-rtmp. [Zugriff am 28. 11. 2022].

[121] A. C. a. Contributors, „Supervisor: A Process Control System,“ 2022. [Online]. Available: http://supervisord.org/. [Zugriff am 28. 11. 2022].

[122] SSH, „SSH (Secure Shell) Home Page,“ SSH, 2022. [Online]. Available: https://www.ssh.com/academy/ssh. [Zugriff am 28. 11. 2022].

[123] N. D. Maintainers, „nginx,“ Docker Inc., 2022. [Online]. Available: https://hub.docker.com/_/nginx. [Zugriff am 29. 11. 2022].

[124] A. C. a. Contributors, „Configuration File,“ supervisord, 2022. [Online]. Available: http://supervisord.org/configuration.html. [Zugriff am 29. 11. 2022].

[125] Microsoft, „File.Move Methode,“ Microsoft, 2022. [Online]. Available: https://learn.microsoft.com/de-de/dotnet/api/system.io.file.move?view=net-7.0. [Zugriff am 19. 12. 2022].

[126] T. Dykstra, C. Ross und S. Halter, „Kestrel web server implementation in ASP.NET Core,“ Microsoft, 23. 07. 2022. [Online]. Available: https://learn.microsoft.com/en-us/aspnet/core/fundamentals/servers/kestrel?view=aspnetcore-7.0. [Zugriff am 09. 12. 2022].

[127] Microsoft, „Single-file deployment and executable,“ Microsoft, 2022. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/core/deploying/single-file/overview?tabs=cli. [Zugriff am 19. 12. 2022].

[128] Microsoft, „ReadyToRun Compilation,“ 2022. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/core/deploying/ready-to-run. [Zugriff am 19. 12. 2022].

[129] Nginx, „nginx,“ 2022. [Online]. Available: https://hub.docker.com/_/nginx. [Zugriff am 08. 12. 2022].

[130] Microsoft, „Install the .NET SDK or the .NET Runtime on Debian,“ Microsoft , 2022. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/core/install/linux-debian. [Zugriff am 20. 12. 2022].

[131] P. Khushalani, „What Is Cloud Computing?,“ in *Kubernetes Application Developer*, Berkeley, CA, Apress, 2022, pp. 1-20.

[132] kubernetes, „Was ist Kubernetes?,“ 2022. [Online]. Available: https://kubernetes.io/de/docs/concepts/overview/what-is-kubernetes/. [Zugriff am 21. 12. 2022].

[133] D. Vohra, „Hello Kubernetes,“ in *Kubernetes Microservices with Docker*, Berkeley, CA, Apress, 2016, p. 39–76.

[134] kubernetes, „Using Minikube to Create a Cluster,“ 2022. [Online]. Available: https://kubernetes.io/docs/tutorials/kubernetes-basics/create-cluster/cluster-intro/. [Zugriff am 21. 12. 2022].

[135] B. Schmeling und M. Dargatz, „The Impact of Kubernetes on Development,“ in *Kubernetes Native Development*, Berkeley, CA, Apress, 2022, pp. 1-57.

[136] Microsoft, „Was ist Azure DevOps?,“ 18. 11. 2022. [Online]. Available: https://learn.microsoft.com/de-de/azure/devops/user-guide/what-is-azure-devops?view=azure-devops. [Zugriff am 05. 01. 2023].

[137] Microsoft, „Was ist Azure Pipelines?,“ Microsoft, 05. 01. 2023. [Online]. Available: https://learn.microsoft.com/de-de/azure/devops/pipelines/get-started/what-is-azure-pipelines?view=azure-devops. [Zugriff am 09. 01. 2023].

[138] T. K. Authors, „Ingress,“ 05. 12. 2022. [Online]. Available: https://kubernetes.io/docs/concepts/services-networking/ingress/. [Zugriff am 09. 01. 2023].

[139] G. Contributors, „ansible,“ 2007. [Online]. Available: https://github.com/ansible/ansible. [Zugriff am 09. 01. 2023].

[140] A. p. contributors, „Getting started with Ansible,“ 14. 12. 2022. [Online]. Available: https://docs.ansible.com/ansible/latest/getting_started/index.html. [Zugriff am 09. 01. 2023].

[141] A. p. contributors, „Installing Ansible,“ 14. 12. 2022. [Online]. Available: https://docs.ansible.com/ansible/latest/installation_guide/intro_installation.html. [Zugriff am 09. 01. 2023].

[142] S. Hegde, „10 Ansible modules you need to know,“ Red Hat, Inc., 11. 09. 2019. [Online]. Available: https://opensource.com/article/19/9/must-know-ansible-modules. [Zugriff am 09. 01. 2023].

[143] A. p. contributors, „Creating a playbook,“ 14. 12. 2022. [Online]. Available: https://docs.ansible.com/ansible/latest/getting_started/get_started_playbook.html. [Zugriff am 09. 01. 2023].

[144] L. Guillen, „10 NSSM (Non-Sucking Service Manager) Best Practices,“ CLIMB, 24. 12. 2022. [Online]. Available: https://climbtheladder.com/10-nssm-non-sucking-service-manager-best-practices/. [Zugriff am 09. 01. 2023].

[145] I. Patterson, „NSSM,“ [Online]. Available: https://nssm.cc/scenarios. [Zugriff am 09. 01. 2023].

[146] I. Patterson, „NSSM - the Non-Sucking Service Manager,“ 2007. [Online]. Available: https://nssm.cc/usage. [Zugriff am 09. 01. 2023].

[147] K. R. Vijayanagar, „HLS Packaging using FFmpeg – Easy Step-by-Step Tutorial,“ 21. 12. 2020. [Online]. Available: https://ottverse.com/hls-packaging-using-ffmpeg-live-vod/. [Zugriff am 08. 11. 2022].

[148] T. V. Community, „Time v3.2,“ 2021. [Online]. Available: https://addons.videolan.org/p/1154032. [Zugriff am 10. 01. 2023].

16. Anhangsverzeichnis

17. Anhang

A. ffmpeg-Skripte

A.1. Skript zur Bildschirmaufnahme (Skript 1)

```
@echo off

cd Streaming-files\ffmpeg
echo Checking desktop-monitor-size...
:: wmic desktopmonitor get screenheight, screenwidth
for /f "tokens=1,2 delims==" %%i in ('wmic desktopmonitor get
screenheight^,screenwidth /value ^| find "="') do (
 if "%%i"=="ScreenHeight" set height=%%j
 if "%%i"=="ScreenWidth" set width=%%j
)
echo your screen is %width% * %height% pixels

::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:: Option 1: gdigrab (should be installed on Windows by default)
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
echo Starting to record desktop monitor with gdigrab...

ffmpeg -f gdigrab -framerate 24 -offset_x 0 -offset_y 0 -video_size
%width%x%height% -i desktop -preset ultrafast -tune zerolatency -sc_threshold
0 -r 24 -g 24 -f flv rtmp://localhost:1935/live/stream1

::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:: Alternative 2: UScreenCapture
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

:: echo checking if UScreenCapture is already installed ...
:: ffmpeg -list_devices true -f dshow -i dummy 2>&1 | findstr "UScreenCapture"
>temp.txt

:: del temp.txt
:: IF errorlevel 1 ( echo Installling UScreenCapture... && cd .. && cd
UScreenCapturex642018 && UScreenCapture_x64.msi && echo Finished!)^
:: ELSE (echo UScreenCapture already installed!)
:: echo Starting to record and stream desktop monitor with UScreenCapture...
:: cd ..
:: cd ffmpeg

:: ffmpeg -f dshow -i video="UScreenCapture" -framerate 30 -rtbufsize 2.14748M
-vf crop=1920:1080 -c:v libx264rgb -crf 0 -preset ultrafast -tune zerolatency
-keyint_min 24 -sc_threshold 0 -r 24 -g 24 -f flv
rtmp://localhost/live/stream1

PAUSE
```

A.2. Skript zur Segmentierung des RTMP-Streams (Skript 2)

```
@echo off

echo Starting to segment stream...

cd Streaming-Files/ffmpeg

ffmpeg -i rtmp://localhost:1935/live/stream1 -preset ultrafast -tune
zerolatency -sc_threshold 0 -r 24 -g 24 -f hls -hls_time 1 -hls_list_size 160
../hls/stream1_.m3u8

:: to delete segments automatically: -hls_flags delete_segments

PAUSE
```

B. C#-REST-API

B.1. SegmentController.cs

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.Net.Http.Headers;
using System.Net;
using System.Reflection;
using System.Timers;

namespace TBM_REST_API.Controllers
{
  [Route("api/[controller]")]
  [ApiController]
  public class SegmentController : ControllerBase
  {
    private static string filePath = @".\hls\";
    // GET: api/segment/all
    [HttpGet("all")]
    public IEnumerable<Segment> Get()
    {
      Console.WriteLine("Called /api/segment/all");
      List<Segment> allSegments = getAllSegments();
      int length = 0;
      if(allSegments.Count > 0){
        length = allSegments.Count -1;
      }
      return Enumerable.Range(0, length).Select(index => new Segment
      {
        fileName = allSegments[index].fileName,
        fileSize = allSegments[index].fileSize,
        sequenceNum = allSegments[index].sequenceNum,
        dateOfCreation = allSegments[index].dateOfCreation
      })
      .ToArray();
    }

    // GET: api/segment/{id} -->get specific .ts File (hls-segment)
    [HttpGet("{id}")]
    public async Task GetHLS_segment(int id)
    {
      Console.WriteLine("Called /api/segment/" + id);
      string hlsSegmentPath = filePath + "stream1_" + id + ".ts";
      try{
        using(FileStream stream = new FileStream(hlsSegmentPath,
           FileMode.Open, FileAccess.Read, FileShare.ReadWrite)){
          Response.StatusCode = (int)HttpStatusCode.OK;
          Response.Headers.Add( HeaderNames.ContentDisposition,
              $"attachment; filename=\"{Guid.NewGuid()}.ts\"" );
          Response.Headers.Add( HeaderNames.ContentType,
                  "application/octet-stream");
```

```
        await stream.CopyToAsync(Response.Body);
        await Response.Body.FlushAsync();
        stream.Close();
      }
    }
    catch(Exception e){
      Console.WriteLine(e);
      Response.StatusCode = (int)HttpStatusCode.BadRequest;
      await Response.Body.FlushAsync();
    }
  }

   // GET: api/segment/playlist
  [HttpGet("playlist")]
  public async Task GetPlaylist()
  {
    Console.WriteLine("Called /api/segment/playlist");
    string playlistFilePath = filePath + "stream1_.m3u8";
    string playlistTempFilePath = filePath + "stream1_temp.m3u8";

    try{
      System.IO.File.Copy(playlistFilePath, playlistTempFilePath, true);
      using(FileStream stream1 = new FileStream(
                  playlistTempFilePath, FileMode.Open,
                  FileAccess.Read, FileShare.ReadWrite))
      {
        Response.StatusCode = (int)HttpStatusCode.OK;
        Response.Headers.Add( HeaderNames.ContentDisposition,
           $"attachment; filename=\"{Guid.NewGuid()}.m3u8\"");
        Response.Headers.Add( HeaderNames.ContentType,
               "application/vnd.apple.mpegurl");
        await stream1.CopyToAsync(Response.Body);
        await Response.Body.FlushAsync();
      }
      System.IO.File.Delete(playlistTempFilePath);
    }
    catch(Exception e){
      Console.WriteLine(e);
      Response.StatusCode = (int)HttpStatusCode.BadRequest;
      await Response.Body.FlushAsync();
    }
  }

  /** Returns a list with all available segments in the folder 'hls' **/
  private List<Segment> getAllSegments()
  {
    string[] list = Directory.GetFiles(filePath, "*.ts",
            SearchOption.AllDirectories);
    List<Segment> segments = new List<Segment>();
    foreach (string file in list)
```

```
    {
      FileInfo info = new FileInfo(file);
      string filename = info.Name; //gets Filename
      long fileSize = info.Length; //gets Filesize in Bytes

      // splits filename to retrieve sequence number
      string[] stringParts1 = file.Split("stream1_");
      string[] stringParts2 = stringParts1[1].Split(".ts");
      int sequenceNumber = int.Parse(stringParts2[0]);

      DateTime dateOfCreation = info.CreationTime;

      Segment seg = new Segment(filename, fileSize, sequenceNumber,
                  dateOfCreation);
      segments.Add(seg);
    }
    return segments;
  }

  public static void deleteOldSegments(){
    System.Timers.Timer timer = new System.Timers.Timer();
    int interval = 2000; // Interval of 2 seconds
    timer.Elapsed += new ElapsedEventHandler(DeleteSegments);
    timer.Interval = interval;
    timer.Start();
    while (Console.Read() != 'q') {; }
  }

  static void DeleteSegments(object source, ElapsedEventArgs e){
    // Console.WriteLine("Checking playlist-length...");
    int maxLength = 165;

    try
    {
      int amountOfFilesInDir = 0;
      DirectoryInfo dirInfo = new DirectoryInfo(filePath);
      foreach(FileInfo fileInfo in dirInfo.GetFiles()){
        amountOfFilesInDir++;
      }

      if(amountOfFilesInDir > maxLength){
        Tuple<int, int> segments =
                getSmallestLargestSegmentNumber(filePath);
        int largestSegment = segments.Item1;
        int smallestSegment = segments.Item2;

        DirectoryInfo directory = new DirectoryInfo(filePath);
        foreach(FileInfo file in directory.GetFiles())
        {
          if(file.Name.StartsWith("stream1_") &&
```

```
            file.Name.EndsWith(".ts")){
            // splits filename to retrieve sequence number
            try{
              string[] stringParts1 =
                       file.Name.Split("stream1_");
              if(stringParts1.Length > 1){
                string[] stringParts2 =
                       stringParts1[1].Split(".ts");
                int sequenceNumber =
                        int.Parse(stringParts2[0]);
                if(sequenceNumber > largestSegment ||
                  sequenceNumber < smallestSegment){
                  try{
                    file.Delete();
                  }
                  catch(Exception exp){
                    Console.WriteLine("Problem
                           deleting file...");
                    Console.WriteLine(exp);
                  }
                }
              }
            }
            catch(Exception exp){
              Console.WriteLine(exp);
            }
          }
        }
      }
    }
    catch(Exception exception)
    {
      Console.WriteLine("Problem deleting files...");
      Console.WriteLine(exception);
    }
  }

  private static Tuple<int, int> getSmallestLargestSegmentNumber(
                  string path)
  {
    Tuple<int, int> largestSmallestNumbers;
    int smallestNumber = 0;
    int largestNumber = 0;
    string filePath = path + "stream1_.m3u8";
    List<string> linesList = new List<string>();
    // Get number of lines in m3u8-playlist-file
    try{
      using(var file = System.IO.File.Open(filePath, FileMode.Open,
               FileAccess.Read, FileShare.ReadWrite))
      {
```

```
        using (StreamReader reader =
            System.IO.File.OpenText(filePath))
        {
          string line = String.Empty;
          while((line = reader.ReadLine()) != null)
          {
            linesList.Add(line);
          }
        }
      }
      int numberOfLines = linesList.Count;

      for(int i = 0; i < numberOfLines; i++){
        if(i == 3){
          string line = linesList[i];
          string[] splitLine =
                    line.Split("#EXT-X-MEDIA-SEQUENCE:");
          smallestNumber = int.Parse(splitLine[1]);
        }
        if(i == (numberOfLines-1) &&
             linesList[i].StartsWith("stream1_")){
          string line = linesList[i];
          string[] splitLine = line.Split("stream1_");
          string[] numberSegment = splitLine[1].Split(".ts");
          largestNumber = int.Parse(numberSegment[0]);
        }
      }
      largestSmallestNumbers = new Tuple<int, int>(largestNumber,
                               smallestNumber);
      return largestSmallestNumbers;
    }
    catch(Exception e){
      Console.WriteLine(e);
    }
    return null;
  }
```

B.2. Segment.cs

```
namespace TBM_REST_API
{
  public class Segment
  {
    public string fileName { get; set; }
    public long fileSize { get; set; }
    public int sequenceNum { get; set; }
    public DateTime dateOfCreation { get; set; }

    public Segment() { }

    public Segment(string filename, long fileSize, int sequenceNum,
            DateTime dateOfCreation)
    {
      this.fileName = filename;
      this.fileSize = fileSize;
      this.sequenceNum = sequenceNum;
      this.dateOfCreation = dateOfCreation;
    }
  }
}
```

C. C#-REST-Client

C.1. Program.cs

```
using System;
using System.Net.Http.Headers;
using System.Net.Http;
using System.Text.Json;
using System.Net;
using System.Text;
using System.Timers;
using System.Collections.Generic;

Console.WriteLine("*********************************************************");
Console.WriteLine("*C#-Rest-Client to download streaming segments (Chunks)*");
Console.WriteLine("*********************************************************");

/****** Set global Variables and call method start() to start the client ****/

// Create HttpClient for Get-Requests
var handler = new HttpClientHandler()
{
  ServerCertificateCustomValidationCallback =
HttpClientHandler.DangerousAcceptAnyServerCertificateValidator
};
var client = new HttpClient(handler);

// get Environment-Variable (ip-address + port)
var envValue = System.Environment.GetEnvironmentVariable("API"); // should be
in the format: https://IP-Address:Portnumber

// set API-Address
string address = envValue.ToString() + "/api/segment/";

// set local Path (to store downloaded playlist & segments)
string localPath = @"\srv\hls\";

// set playlist-filepaths
string finalPlaylist = localPath + "stream1_.m3u8"; // final file that is
                            called by the Browser
string tempPlaylist = localPath + "stream1_temp.m3u8"; // temporary file that
                           new lines are written to
string apiPlaylist = localPath + "stream1_api.m3u8"; // temporary playlist
        // from the api (only used to determine lowest/highest segments)

// List with segment-numbers (in playlist)
List<int> currentSegments = new List<int>();
int playlistLength = 20; // Max-Playlist-Length
```

```
// set segment-variables
int firstSegmentNr = 0;
int sequenceCounter = 0;
int recentNumber = 0;

// Create Timer-object
System.Timers.Timer newTimer;

// move old files
moveOldFiles();

// call start() to start the Client
start();

/****************** Start / Restart Downloads ************************/
void start()
{
  // 1: Reset Segment-Variables
  firstSegmentNr = 0;
  sequenceCounter = 0;
  newTimer = new System.Timers.Timer();

  // 2: delete all segments from currentSegments
  currentSegments.Clear();

  // 3: Download playlist to determine most recent segment-number
  Task.WaitAll(DownloadPlaylist());

  // 4: Set sequenceNumber (based on recentNumber determined by
              DownloadPlaylist())
  if (recentNumber > 0)
  {
    firstSegmentNr = recentNumber - 1; // recentNumber -1 because last
                segments may still be in process of writing
  }
  else
  {
    firstSegmentNr = recentNumber; // if recentNumber == 0 there is no
                              previous file
  }
  sequenceCounter = firstSegmentNr;
  Console.WriteLine("First requested segment: " + firstSegmentNr);

  // 5: Start timer (which downloads segments & updates playlist regularly)
  startTimer();
}
```

```
/************ Move all existing segments on server to new folder ************/
void moveOldFiles()
{
  DirectoryInfo directory = new DirectoryInfo(localPath);
  string[] files = System.IO.Directory.GetFiles(localPath);
  // if files exist, move them to new folder
  if (files.Length != 0)
  {
    string time = DateTime.Now.ToString();
    Console.WriteLine(time);
    StringBuilder folderNameBuilder = new StringBuilder();
    string newFolder = ""; // build new folder-name based on current
                 date+time
    for (int i = 0; i < time.Length; i++)
    {
      if (time[i].Equals(' ') || time[i].Equals(':'))
      {
        folderNameBuilder.Append("-");
      }
      else
      {
        folderNameBuilder.Append(time[i].ToString());
      }
      newFolder = folderNameBuilder.ToString();
    }
    try
    {
      string newFolderPath = localPath + newFolder;
      // Create new Folder
      System.IO.Directory.CreateDirectory(newFolderPath);
      // Move all files to new Folder
      Console.WriteLine();
      Console.WriteLine("Moving all old segments to new folder...");
      foreach (FileInfo file in directory.GetFiles())
      {
        string sourceFile = localPath + file.Name;
        string destinationFile = newFolderPath + @"\" + file.Name;
        System.IO.File.Move(sourceFile, destinationFile, true);
      }
      Console.WriteLine("Complete.");
    }
    catch (Exception e)
    {
      Console.WriteLine("Problem moving files to folder...");
      Console.WriteLine(e);
    }
  }
}
```

```
/*********** Call API-endpoint to download playlist ***********************/
async Task DownloadPlaylist()
{
  string endpoint = address + "playlist";
  Uri restUri = new Uri(endpoint);
  Console.WriteLine("Downloading playlist ...");
  try
  {
    var response = await client.GetAsync(restUri);
    if (response.IsSuccessStatusCode)
    {
      try
      {
        if (File.Exists(apiPlaylist))
        {
          File.Delete(apiPlaylist);
        }
        using (var fs = new FileStream(apiPlaylist, FileMode.Create))
        {
          await response.Content.CopyToAsync(fs);
        }
        if (File.Exists(apiPlaylist))
        {
          List<string> linesList =
                   File.ReadAllLines(apiPlaylist).ToList();
          recentNumber = getLargestSegmentID(linesList);
          Console.WriteLine("Success!");
          File.Delete(apiPlaylist);
        }
        else
        {
          Console.WriteLine(response.StatusCode);
          Console.WriteLine("Failed to download playlist...
                    Retrying...");
          // stop and dispose timer + restart the client by calling
            start()
          newTimer.Stop();
          newTimer.Dispose();
          start();
        }
      }
      catch (Exception e)
      {
        Console.WriteLine(e);
      }
    }
    else
    {
      Console.WriteLine(response.StatusCode);
      Console.WriteLine("Failed to download playlist... Retrying...");
```

```
      // stop and dispose timer + restart the client by calling start()
      newTimer.Stop();
      newTimer.Dispose();
      start();
    }
  }
  catch (Exception e)
  {
    Console.WriteLine("Failed to download playlist... Retrying...");
    Console.WriteLine(e);
    newTimer.Stop();
    newTimer.Dispose();
    start();
  }
}

/***************** Start timer with interval of 1 second ******************/
void startTimer()
{
  int interval = 1000; // Interval of 1 second
  newTimer.Elapsed += new ElapsedEventHandler(DownloadSegmentAtTimeEvent);
  newTimer.Interval = interval;
  newTimer.Start();

  while (Console.Read() != 'q') {; }
}

/****** Call method to download next Segment once timer is 0 ***************/
void DownloadSegmentAtTimeEvent(object source, ElapsedEventArgs e)
{
  Console.WriteLine();
  Console.WriteLine(DateTime.Now);
  Task.WaitAll(downloadAndUpdate());
  sequenceCounter++; // increase segment-number by 1
}
/****** Call methods to download next Segment and update playlist file ******/
async Task downloadAndUpdate()
{
  Task.WaitAll(DownloadSegment(sequenceCounter)); // wait for segment to be
                              downloaded
  Task.WaitAll(UpdatePlaylist(sequenceCounter));
  Boolean isValid = ValidPlaylist();
  Console.WriteLine("Validating playlist...");
  if (!isValid)
  {
    Console.WriteLine("Problem with playlist or files... either files are
              missing or playlist is invalid.");
    Console.WriteLine("Retrying...");
    // stop and dispose timer + restart the client by calling start()
    newTimer.Stop();
```

```
    newTimer.Dispose();
    start();
  }
  else
  {
    Console.WriteLine("Valid playlist!");
  }
  // Copy / Move temporary playlist to finalPlaylist
  try
  {
    if (!File.Exists(tempPlaylist))
    {
      using (FileStream fs = File.Create(tempPlaylist)) { }
    }
    // Move the file
    File.Move(tempPlaylist, finalPlaylist, true);
  }
  catch (Exception e)
  {
    Console.WriteLine("The copy-process failed: {0}", e.ToString());
  }
}
/*************** Call API-endpoint to download segment **********************/
async Task DownloadSegment(int id)
{
  string fileName = "stream1_" + id.ToString() + ".ts";
  string endpoint = address + id.ToString();
  string localTarget = localPath + fileName;

  Uri restUri = new Uri(endpoint);
  Console.WriteLine("Downloading " + fileName + " ...");

  try
  {
    var response = await client.GetAsync(restUri);
    if (response.IsSuccessStatusCode)
    {
      using (var fs = new FileStream(localTarget, FileMode.OpenOrCreate,
                FileAccess.ReadWrite, FileShare.ReadWrite))
      {
        await response.Content.CopyToAsync(fs);
      }
      if (File.Exists(localTarget))
      {
        Console.WriteLine("Success!");
      }
      else
      {
        Console.WriteLine("Problem downloading segment... file does
                  not exist.");
```

```
        newTimer.Stop();
        newTimer.Dispose();
        start();
      }
    }
    else
    {
      Console.WriteLine(response.StatusCode);
      Console.WriteLine("Failed to download segment... Retrying...");
      // stop and dispose timer + restart the client by calling start()
      newTimer.Stop();
      newTimer.Dispose();
      start();
    }
  }
  catch (Exception e)
  {
    Console.WriteLine("Failed to download segment... Retrying...");
    Console.WriteLine(e);
    // stop and dispose timer + restart the client by calling start()
    newTimer.Stop();
    newTimer.Dispose();
    start();
  }
}
/************** Rewrite playlist (adds downloaded segments) ****************/
async Task UpdatePlaylist(int segmentID)
{
  currentSegments.Add(segmentID);

  // now delete extra segments from original
  if (currentSegments.Count > playlistLength)
  {
    // remove first segment (since playlist is always updated after each
      added segment)
    currentSegments.RemoveAt(0);
  }
  try
  {
    // File.Delete(tempPlaylist);
    using (var file = File.Open(tempPlaylist, FileMode.Create,
             FileAccess.ReadWrite, FileShare.Read))
    {
      file.Seek(0, SeekOrigin.End);
      using (var stream = new StreamWriter(file))
      {
        if(currentSegments.Count > 0)
        {
```

```
          stream.WriteLine("#EXTM3U\r\n#EXT-X-VERSION:3\r\n#EXT-X-
TARGETDURATION:1\n#EXT-X-MEDIA-SEQUENCE:" + currentSegments[0] +
"\r\n#EXTINF:1.000000,\r\nstream1_" + currentSegments[0] + ".ts");
          for (int i = 1; i < currentSegments.Count; i++)
          {
            int currentsegmentID = currentSegments[i];
            stream.WriteLine("#EXTINF:1.000000,\r\nstream1_" +
                     currentsegmentID + ".ts");
          }
        }
      }
    }
  }
  catch (Exception e)
  {
    Console.WriteLine("Problem updating playlist...");
    Console.WriteLine(e);
  }
}
/************************* Validating playlist ****************************/
Boolean ValidPlaylist()
{
  Boolean isValid = true;
  int previousNumber = 0;
  for (int counter = 0; counter < currentSegments.Count; counter++)
  {
    // check if file with segmentNr = number exists
    int number = currentSegments[counter];
    string currentFileName = "stream1_" + number + ".ts";
    string path = localPath + currentFileName;
    // if the file does not exist, then return isValid = false
    if (!File.Exists(path))
    {
      isValid = false;
      return isValid;
    }
    // else file exists: check if sequence numbers are in ascending order
    else
    {
      if (counter == 0)
      {
        previousNumber = number;
      }
      else
      {
        if (number == (previousNumber + 1))
        {
          previousNumber = number;
        }
        else
```

```
        {
          isValid = false;
          return isValid;
        }
      }
    }
  }
  return isValid;
}
/****** Get Largest Segment-ID from list of all lines in playlist file ******/
static int getLargestSegmentID(List<string> allLines)
{
  int largestSegment = 0;
  int counter = 0;
  foreach (var line in allLines)
  {
    if (line.StartsWith("stream1_") && line.EndsWith(".ts"))
    {
      // determine sequence number of this segment & check if it is in
        correct sequence
      try
      {
        string[] splitLine = line.Split("stream1_");
        string[] numberSegment = splitLine[1].Split(".ts");
        int sequenceNumber = int.Parse(numberSegment[0]);
        if (counter == 0)
        {
          largestSegment = sequenceNumber;
        }
        else
        {
          if (sequenceNumber > largestSegment)
          {
            largestSegment = sequenceNumber;
          }
        }
      }
      catch (Exception e)
      {
        Console.WriteLine(e);
      }
      counter++;
    }
  }
  return largestSegment;
}
```

C.2. Segment.cs

```
using System.Text.Json.Serialization;

public record class Segment(
  [property: JsonPropertyName("fileName")] string Name,
  [property: JsonPropertyName("fileSize")] long FileSize,
  [property: JsonPropertyName("sequenceNum")] int SequenceNumber,
  [property: JsonPropertyName("dateOfCreation")] DateTime DateOfCreationUtc)
  {
    public DateTime DateOfCreation => DateOfCreationUtc.ToLocalTime();
  }
```

D. Kubernetes-Konfiguration (streaming-test.yml)

```
apiVersion: apps/v1
kind: Deployment
metadata:
 name: streaming-test
 namespace: herrenknecht-connected
spec:
 replicas: 1
 selector:
  matchLabels:
   app: streaming-test
 strategy:
  type: Recreate
 template:
  metadata:
   labels:
    app: streaming-test
  spec:
   nodeSelector:
    "kubernetes.io/os": linux
   containers:
    - name: streaming-test
     image: image-path
     imagePullPolicy: IfNotPresent
     resources:
      requests:
       memory: "256Mi"
      limits:
       ephemeral-storage: 256Mi
       memory: "512Mi"
     ports:
      - containerPort: 80
     env:
      - name: API
       value: "path/ip-address:port”
   terminationGracePeriodSeconds: 5
---
apiVersion: v1
kind: Service
metadata:
 name: streaming-test
 namespace: herrenknecht-connected
spec:
 selector:
  app: streaming-test
 ports:
  - protocol: TCP
```

```
   port: 80
   targetPort: 80
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
 annotations:
  kubernetes.io/ingress.class: nginx
  nginx.ingress.kubernetes.io/proxy-buffer-size: 128k
  nginx.ingress.kubernetes.io/rewrite-target: /$2
 name: streaming-test
 namespace: herrenknecht-connected
spec:
 rules:
  - host: host-url
   http:
    paths:
     - backend:
       service:
        name: streaming-test
        port:
         number: 80
      path: /streaming-test(/|$)(.*)
      pathType: Prefix
 tls:
  - hosts:
    - host-url
   secretName: secret-url
```

E. Latenzmessungen

E.1. Ergebnis 1: lokal ohne Rest-API & -Client

Aufnahme	Datum	Bildschirm (in s)	Stream (in s)	Latenz (in s)
1	10.01.23	12,40	16,20	3,80
2	10.01.23	9,90	13,65	3,75
3	11.01.23	11,15	15,90	4,75
4	11.01.23	9,15	13,15	4,00
5	11.01.23	9,40	13,15	3,75
6	12.01.23	13,15	17,90	4,75
7	12.01.23	5,15	9,40	4,25
8	12.01.23	6,40	11,15	4,75
9	12.01.23	8,40	13,40	5,00
10	13.01.23	11,15	15,40	4,25
11	13.01.23	6,40	10,90	4,50
12	13.01.23	7,65	12,65	5,00
13	13.01.23	13,40	18,15	4,75
14	16.01.23	9,90	14,15	4,25
15	16.01.23	6,40	10,15	3,75
16	16.01.23	7,40	11,40	4,00
			Total	**69,30**
			Durchschnitt	**4,33**
			Minimum	**3,75**
			Maximum	**5,00**

E.2. Ergebnis 2: Testrechner mit Rest-API & -Client

Aufnahme	Datum	Bildschirm (in s)	Stream (in s)	Latenz (in s)
1	10.01.23	13,40	20,15	6,75
2	10.01.23	4,90	12,40	7,50
3	11.01.23	42,65	49,90	7,25
4	11.01.23	8,90	16,15	7,25
5	11.01.23	17,48	24,24	6,76
6	12.01.23	11,65	19,15	7,50
7	12.01.23	9,65	17,23	7,58
8	12.01.23	8,40	16,15	7,75
9	12.01.23	8,40	15,65	7,25
10	13.01.23	9,65	16,90	7,25
11	13.01.23	6,40	12,65	6,25
12	13.01.23	6,40	13,65	7,25
13	13.01.23	8,15	15,65	7,50
14	16.01.23	9,15	14,40	5,25
15	16.01.23	7,40	14,40	7,00
16	16.01.23	5,65	13,15	7,50
			Total	**113,59**
			Durchschnitt	**7,10**
			Minimum	**5,25**
			Maximum	**7,75**